Technical University of Denmark

Beer Recipe Calculation system

Bachelor Project

Author:
Dan Roland Persson (s134837)

Project-instructor:
Christian D. Jensen

Jan 3rd 2017

Contents

1 Executive Summary

The goal of this report is to describe the development of a web-based brew recipe designer and the system created around this designer. The goal of the overall project being to create a useful tool for creating beers. The project itself was carried out as a bachelor thesis (or project) in fall of 2016 and was handed in Jan 3rd 2017. The project was first pitched by Associate Professor Christian Damsgaard Jensen as a bachelor thesis and was in turn planned, designed and executed by student Dan Roland Persson.

The project consists mainly of software engineering, mixed with more general project management and is comprised of creating a full software system. This including development of a database, application layer, testing and design of a GUI. In addition to this several none-functional requirements were considered and integrated into the project, such as database redundancy, expandability, maintainability, risks associated to the project and some aspects of security. Lastly several development methodologies are reviewed and the used methodologies impact upon the project is discussed.

Overall the project can be declared a success, per the stakeholder satisfaction and completion of the vision, supported by use cases. The material presented in most courses at DTU are most often due to time constraints disconnected from one another. Therefore, a larger scale project provides a good chance to create a full-fledged software system. In this case the software's functionality is analysed, designed and implemented to help brewers create beer recipes.

2 Introduction

Brewing has taken place since ancient times, however it has not always been possible to predict characteristics of brewing, thus leaving the results less than predictable. In more modern times and with the prevalence of modern understandings of chemistry it has become increasingly possible to predict the outcome of a brewing process, although taste remains impossible to predict. It is however possible to mathematically predict color, bitterness, alcohol and other aspects of potential brew.

In addition to this, the increasing use of computers allows the task of predicting these aspects to be more easily achieved. The goal of the project is to create a beer recipe calculation application with a suitable web based user interface. Further a usable system should be designed around this "recipe designer", including functionality such as users, comments, logins and the logical extensions of these.

The database is to be designed with thoughts of future extensions of functionality and be structured in such away as to minimize redundancy. This report will follow a structure that

- Explores the theory of brewing and core technologies;
- Analyses the basics of this project;
- Lists some considerations made during the design-phase of the application;
- Goes in detail with the implementation of the application;
- Analyses the results from testing the application; and
- Discusses, and concludes, the correctness of the results, and if they fit the given requirements.

The main content of the report will only contain small and/or parts of larger figures. To keep the report structured, and the page content reasonable, all the larger figures and illustrations will be stored in the appendix, and referred to in the relevant sections.

2.1 Installation of application on local machine

The project itself requires a MYSQL database setup with the default settings accompanied by the MYSQL workbench, the database should be setup with the root account having the password "tukanlobo19". Once the database is running, the sql files; creation and population can be run in that order. Once completed the database should be ready and populated with the default data. Finally the project can be deployed though an IDE such as eclipse, with a tomcat 8 server installed (can be found on the Apache web site as a zip). If the project contains errors after the import, simply deploy it on the tomcat 8 server. In addition the project may need to be run on server twice after linking the tomcat server to eclipse. The project is then usable from the address "http://localhost:8080/BrewRecipeSystem/" or through eclipses embedded browser. In addition to this an Admin user has been prepared in the default data, with the name "DanRoland" and password "Password1". It should be noted the system has been mostly tested in Chrome, although Firefox has been tested somewhat as well.

3 Theory

In order to get a better overview of the requirements of the recipe designer and of the general system we must explore some of the theory behind brewing and of the core technologies we are to make use of. The brewing theory may also provide an overview of the data needed to be stored.

3.1 Brewing

For the explanation of individual key words, see the glossary.

For home brewing, one requires a series of ingredients, for example: malt extract, speciality grains, yeast, and hops. Furthermore, the process does require some speciality equipment to brew a batch of beer. While the equipment varies, the process remains the same, firstly we will go through extract brewing and assume a reasonable home brewing batch size. Secondly we will look at all-grain brewing with a mash.
First step of creating a brew requires the steeping of grains in hot water, making sure not to burn them. This process can for example take 30 minutes. At the end of the process one wants to rinse the grains with more hot water to extract any remaining flavouring. Since steeping the grains do not result in enough sugars for a good brew/fermentation, adding malt sugar is important, these can be both a thick syrup or a dried compound. In case of syrup it is important to stir the pot very thoroughly, in case of the dry extract poor it more quickly and stir to dissolve. Once this stage is completed the pot of mixture is referred to as the "wort". Most commonly it is at this point a boil begins (for example 1 hour or more). During this time hops are added periodically, to counter the sweetness of the wort, and to add flavour and aroma. Its possible that the recipe requires multiple different hops added at different points during the boil, this greatly adds to the flavour complexity of the brew.
During the final 15 minutes of the boil, the brewer can also add a clearing agent to the boil, to ensure a clear looking end brew. Once the brew boil is complete its important to quickly cool the pot containing the wort. Most home brewers use a bathtub with ice, however more effective methods exist, if one can afford the equipment. After cooling and reaching a suitable temperature, the wort should be transferred to a container in which the fermentation can happen. At this stage the yeast must be added, once more different versions exist, some versions require for the yeast to be started before being added to the wort. During this stage more water may be added to bring the batch to a proper size for fermentation, once the yeast is added, a airtight lit should be used, furthermore it should be possible for the co2 to escape the container as such one can use an airlock to let the co2 escape. The fermentation step itself may take several weeks and require the brew to the transferred from one fermentation container to another cleaner fermentation container.
When preparing to bottle, it is important to add a priming solution to the brew in order for a controlled fermentation to take place in the bottled brew. This has the function of carbonating the brew.

The brew is then carefully bottled and sealed, and kept in room temperature for approximately 2 weeks, although this time can vary.
During each step of the process its important that all equipment that touches the brew is well sanitised, to ensure no bad bacteria are added to the batch.
For this example its assumed the ingredients are added to the pot in a muzzling bag as to more easily remove ingredients from the pot and further assumes the brew is an extract brew and not a all-grain one. The all grain brews requires a bit more work and preparation as the mashing step requires you to convert the starches from the grain into sugars and furthermore requires some filtering (if muzzling bags are not used) before getting to the boiling step.
Specifically the difference between mash and extract brewing, is the preparation of the grain and extraction of sugar. While extract brewing follows nicely the previously mentioned methodology. Mash on the other hand requires you to process the grain in order to extract the sugars, rather than just steeping some grain and adding an extract.
This method adds a few more steps to the brewing process, simply speaking you heat a batch of water and allows the grains to "rest" in this heated water for a period of time. Given the time and temperature, different levels of sugars can be extracted from the grain. During this stage it is important that the grain is crushed. Furthermore it is important to keep a consistent temperature. Once it has sat for the desired amount of time, it is possible to do a second mashing step here you "wash" the grains of any remaining flavouring and sugars by pouring water into for example a bucket containing the grains removed from the pot. This is also refereed to as second runnings. Once the grains are removed, the second runnings can be added to the first ones and the brewing method proceeds as described in extract brewing. As such one continues from the point of adding hops and boiling the wort.

3.2 The calculations of beer brewing

While it is easily possible to do calculations in regards to a proposed brew, predicting the resulting flavor is not. On the other hand several aspects of prediction are possible such as predicting the bitterness, color and alcohol content, all of which are important in predicting how a batch of brew turns out. Additionally this also allows the brewer to shoot more accurately for a type of brew such as a pilsner for example. The following aspect of a brew can be calculated:

- Original gravity
- Final Gravity
- Alcohol content
- Beer color
- IBU/Hop bitterness
- Calories

3.2.1 Gravity

Usually water would have a gravity of around 1.000, if something is added to the water it becomes more dense, in brewing this means that when adding sugars the gravity will increase and as such can be measured. The formula is:

$$OG = AmountExtract * PPG / batchsize[1] \tag{1}$$

Where ppg is points per pound per gallon for an extract, assuming all sugars are extracted. When dealing with multiple extracts for example this mean they all add some sugar to the water. As such they can be calculated separately and the total gravity points is the total of the separate Original gravities. When dealing with mash, one also needs to consider the efficiency of the mash. The efficiency is determined by the amount of sugar that can be extracted from a grain type, however this is also effected by the equipment available to the brewer. In a case where the efficiency is needed to be taken into account the formula is:

$$OG = AmountExtract * PPG * Efficiency / batchsize[1] \tag{2}$$

The final gravity can be measured once the fermentation is completed, as during the fermentation the yeast will have eaten sugar in the wort and created alcohol and carbon dioxide. As the sugar disappears from the wort, this means there is a different density than with the original gravity. The amount of gravity that disappears is refered to as attenuation. This percent signifies how much sugar the yeast will consume. The attenuation can be calculated via:

$$Attenuation = ((OG - FG)/(OG - 1)) * 100[1] \tag{3}$$

Where OG and FG refer to the original gravity and the final gravity respectively. Most of the time a yeast manufacturer will print the expected attenuation and as such it does not require a test batch to determine. When the attenuation is found or calculated the final gravity can be calculated:

$$FG = 1 + ((TotalGravityPoints * (1 - AttenuationPercent))/1000)[1] \tag{4}$$

The final gravity is the specific gravity once fermentation is completed.

3.2.2 Alcohol content

With the original gravity and final gravity calculated it now becomes possible to calculate the expected alcohol content. During the fermentation process carbon dioxide bubble's out of the airlock while the alcohol stays behind. for each gram of carbon dioxide that leaves the fermenter, approximately 1.05g of alcohol are left behind. At this point the alcohol calculations can be made, however there are two main ways of measuring the alcohol content, either alcohol by weight or alcohol by volume. Most brews use the alcohol by volume measurement as the default. The conversion between these are not difficult. In fact is just requires dividing ABW by the density of ethyl alcohol.

$$ABW = ((OG - FG) * 1.05)/FG[1] \tag{5}$$

$$ABV = (((OG - FG) * 1.05)/FG)/0.79[1] \tag{6}$$

Where 0.79 is the before mentioned density of ethyl alcohol.

3.2.3 SRM/Brew color

The standard way of measuring brew color is with the standard reference method. Degrees lovibond is a way of measurement for the color malts add to the brew. The first thing required for finding the color of a recipe is to calculate the malt color units.

$$MCU = (GrainColor * GrainWeight)/Volume[1] \tag{7}$$

Where grain color is each grains respective lovibond degrees, grain weight in lbs and volume in gallons. Furthermore if the presence of more then one fermentable is used the MCU color is calculated for each and added together. Lastly since light absorbance is logarithmic and not linear, we must use the Morey equation to calculate the final SRM Color.

$$SRMcolor = 1.49 * (MCU * *0.69)[1] \tag{8}$$

Lighter brews tend to have lower SRM numbers, while darker values have higher numbers, additionally any value over 50 is considered black. Finally with the color calculations it should be noted that these are more prone to containing errors, as its effected by boil time, caramelization and other aspects of the brewing process. This means the process is merely an estimate for the color as these aspects can be hard to predict.

3.2.4 IBU/Bitterness

The bitterness of a brew is measured by International bitterness units (IBU). One IBU is the same as one mg of alpha acid per liter of home brew. Determining the IBU is however the most difficult aspect of the brews characteristics, as there exists multiple formula's for doing so, each creating different results. For simplicity we will use the derived calculation method found in Ray Daniel's book "Designing Great Beers"[2]:

$$IBU = (U\% * Alpha\% * Woz * 0.7489)/(Vgal * Cgrav) \tag{9}$$

Where U% is the hop utilization in percent, alpha% is the percent alpha for the hop, W is the hops weight in ounces, and Vgal is the final volume of the wort in gallons and where cgrav is the correction for worts with a gravity above 1.05 during the boil. Yet again if multiple hops exists, one calculates each ones bitterness separately and adds them together for the final result.

3.2.5 Calories

The calories in most brews can be seen as the calories from the alcohol and those from carbs (mostly from the sugar). As such the total calories is the sum of both calculations. The formulas for calorie calculations:

$$CalsAlco = 1881.22 * FG * (OG * FG)/(1.775 - OG)[1] \tag{10}$$

$$CalsCarb = 3550.0 * FG * ((0.1808 * OG) + (0.8192 * FG) - 1.0004)[1] \tag{11}$$

$$Total = CalsAlco + CalsCarb \tag{12}$$

3.3 Database design

Good database design relies on well designed tables. As such the tables should have little to no redundancy. Furthermore it might have proper indexing for the sake of optimisation and have consistent information. Lastly it should also be expandable.
The most common way to ensure good database design is by normalising the database, to reduce redundancy and improve data integrity. As such normalization can help us avoid two of the before mentioned issues. Normalization has several normal forms which have rules that for example ensures elements are atomic and that every Non Primary Key Attribute depends on the entire Primary Key. But even with normalization we are left with two remaining possible pitfalls, namely good indexing and making the system easily expandable.
The issue of making a database expandable relies on having logically structured it. Such that in case a brew should require another attribute, it is a simple matter of adding a column to the existing table. This however does not ensure the addition is in line with the normalized database design. This means some care has to be taken before adding a new column to the table.
On the other hand making proper indexing for the sake of performance is a bit more difficult, one could use views to this matter. At the same time it could just depend on using good primary keys, an example of this might be if 5 types of yeast with different attributes have the same name, this would of course mean all primary keys would be the same, which is impossible, due to them not being unique. The solution to this would perhaps be creating a new unique ID. However this would make searching more difficult for users and perhaps use a lot more space then otherwise needed. Two different possible solutions exist, namely to force the system to create unique names for each of the 5 yeasts (Ex Yeast type A) or expand the primary key to have 2 attributes, thus including an identifier for the differences in the yeasts. From a users perspective, finding the correct yeast by name only would of course be preferable, as otherwise the user might have to read through multiple rows and columns.

However one more aspect of database design is worth considering, namely handling large amounts of data within the system. But what exactly is the difference between bad table design and good table design in this case?

The good design as mentioned earlier is to avoid redundancy in the brew by hav-

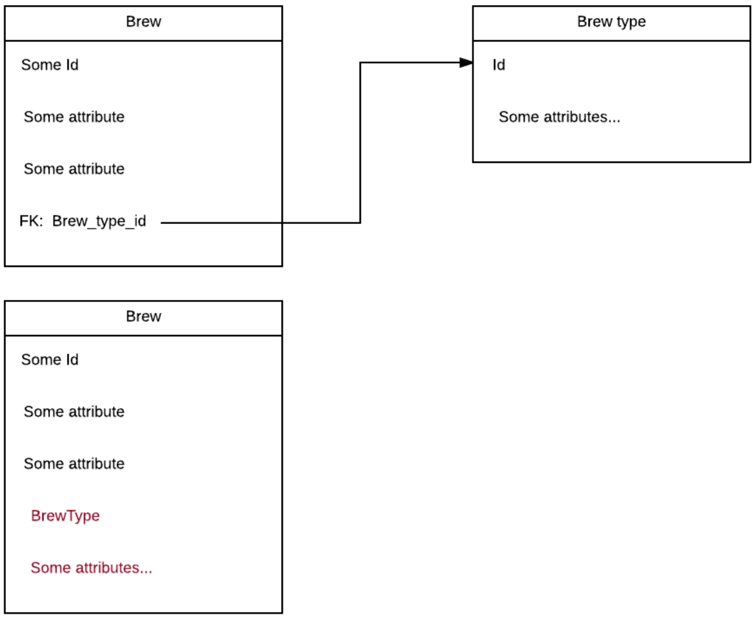


Figure 1: A good and bad table design

ing a separate table for the brew type. Thus reducing the redundancy within the two tables. However during a database lecture with a guest speaker, an interesting point was raised in regard to database design. Namely he argued that there is a difference between good practice and practicality. The general idea was that, one might want to revert away from more complex database structures, even if it is considered good practice. Looking at figure 1 we see two different proposed table solutions. The first is considered "good practice", the second from a normalization stand point "bad". The argument presented for the good practice was if one had to update the brew type, one simply changed the type tables value and all brews would automatically be updated. Why would one instead choose to update hundreds if not millions of brews just to change one type? The counter he used was; if ever you need to retrieve all brews in the table, as well as their brew type, you would need to join to potentially millions of brews with million of types. To explain this argument quickly, the join operation of tables is while powerful and effective, also costly performance wise. Iterating through millions of rows in linear time (or however fast indexing allows for) is acceptable, but having to join each row with a relational table is not.

This does however not mean one should disregard the good practises of relational databases, as this would only be applicable in some special cases. For this project is seems unlikely that the application would ever reach amounts of data where this would be usable. However if a system should be highly scalable these points are well worth considering.

3.4 Security

For a web-based system several security threats are possible, both for the database and the web-front end. For the database specifically, it might be a SQL injection. As for the Login system it might be something like Brute force, session hijacking or Cross-site scripting, among others. As such the system must be able to handle these common types of attacks. One way to deal with SQL injections in java is to simply use prepared statements, so that pitfall is somewhat easily avoided. Cross-site scripting can for example be countered via limiting the input size of text fields such that they cannot insert large segments of script into a text field, additionally it should not be allowed to enter things such as brackets among other special characters, this can be done by sanitising the text data via a filter, fortunately java contains several existing libraries for this purpose. In the database specifically it is important to securely store things such as login name and the users passwords. An important step in creating a secure login would of course be to implement cryptographic measures to the handling of passwords. The most common way of securing the costumers login credentials would be by salting and hashing. When taking a string, like a password or login name, you apply a one-way level of encryption to it, simply put one modulates the string into an unreadable sequence of numbers of characters.

- md5("password") = 5f4dcc3b5aa765d61d8327deb882cf99
- md5("Password") = dc647eb65e6711e155375218212b3964

The example illustrates the hashing of two similar (though different) strings, using md5[1]. While the passwords in plain text are not that far apart, the hashing produces wildly different results. However if the password hashing is a one way deal then how do we match incoming login credentials?

We will never have a need for reversing the process since there are no random elements in the hashing algorithms, if a user tries to login then we simply match the stored hash values with the newly created hash of received data.

However this still leaves the system vulnerable. The vulnerability is namely that it is easy to calculate a possible hash, after all using an existing hash allows others to use it as well. If someone was to bypass this security measure it is possible to do so with a so called "lookup table". A lookup table contains a list of the most common passwords and their hashed versions. This allows the intruder to simply match stored hashes with their passwords. This could be a problem as many users have a tendency to use known passwords. For example the two most common passwords used in 2015 is the passwords:

[1]http://www.md5.cz/

"123456" and "password". Furthermore 6 out of 10 of the most used passwords are simple variations of the numbers 1-10![2] While it is possible to force users to avoid using the most common passwords by creating a set of criteria for every password, this in it self is not enough either, (thought it makes bruteforce attacks harder). A solution to this problem is salting. When salting something you add a random sequence of numbers/characters at the start, end or both of the value being hashed. This in effect will render lookup tables useless.

- md5("password + 1356") = dbbfd73e50ccaf27e41a312c32efa8e4
- md5("password + 1186") = 0131509ca7f2001438a4c18a8490de61

Of course if salting is being used, one would need to store the salts in a table, this is the only way of matching passwords when a user logs in. Still if an intruder were to access the database and collect the salts, they would still need to be applied to the lookup table with every possible hash available. Lastly one must consider the appropriate place to use hashing and salting. Usually it would be considered good practice to both hash at application level and server side to disable intruders from intercepting a password before it has been hashed and salted. A good thing about this type of solution is that many of the best hashes are available online at no cost, so this in it self should not complicate the workload to much. Brute force guessing of passwords is however still possible, given that many of the most used passwords are the same. Not allowing users to use common passwords is one way of strengthening passwords. Another solution is to limit the number of login attempts for a user on a given IP or account. This would however also require that any potential attacker knows the login name.

3.5 Web technologies

Web application has in recent years become ever increasingly used, perhaps due to the ease at which they can be accessed. By definition web applications are distributed applications, in effect meaning they are programs run on more than one computer and communicate through a network or server. These web applications can be accessed via a browser as a user client. Another key advantage of web applications is the ease of which updates can be applied, that is to say the developer has no need to deploy and install these updates of potentially large amounts of systems. In the case of this project, which is defined to be coded in java by the project description (see appendix D), we may make use of javas web application development tools. These are in turn the Java Servlet API and JavaServer Pages(JSP)[3]. The servlets allow us to define http specific classes. The servlet class extending the abilities of the servers that host the applications accessed via request-response programming. Requests and response can be seen as input and output respectively. Requests here most often consist of items inside html $\langle Form \rangle$ tags in the applications server pages. The java server pages are text-based documents that contain two types of text. Static text such as HTML, WML or XML and JSP java elements,

[2]http://gizmodo.com/the-25-most-popular-passwords-of-2015-were-all-such-id-1753591514

which determines how the page constructs dynamic content, which can be displayed in HTML as well. The server pages may also make use of both CSS, javascript, other java classes or even other JSPs to increase functionality and the overall quality of the pages. Finally http or the Hypertext Transfer Protocol is a request-response protocol in client-server computing. It is also one of the most widely used protocols on the internet.

4 Analysis

From the project description we can analyse the project and break it down into use cases and features. The project description can be found in appendix D.

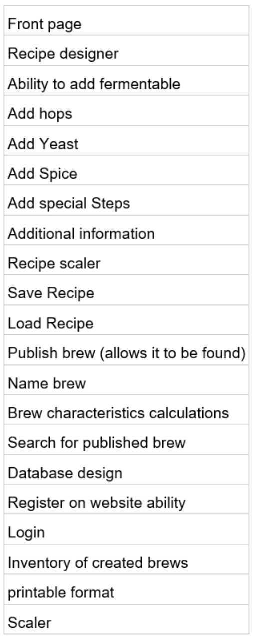
4.1 Problem analysis

As mentioned in the project description, the system should be capable of calculating different aspects of a potential homebrew. Furthermore it should also allow for recipe scaling and printing as these are features a user might need. If a user wishes to make a bigger batch instead of having to up each ingredient manually, the system should be able to scale the required ingredients. At the same time it might be cumbersome for the user of the system to be required to have a laptop close at hand when doing the actual brewing, as such it would be useful for the user to be able to print recipes. Furthermore the system should allow for saving recipes, and publishing them for others to see. The publishing functionality would of course require some sort of relevant search functionality, allowing users to find already created recipes.

Lastly the system should contain the infrastructure needed for it to function, database, application layer, and these should be designed with thoughts of expandability and security respectively. The application should also be tested well for correctness of the application.

4.2 Describing the requirements

With more knowledge about the systems requirements, we now more or less have a required feature listing:



Front page
Recipe designer
Ability to add fermentable
Add hops
Add Yeast
Add Spice
Add special Steps
Additional information
Recipe scaler
Save Recipe
Load Recipe
Publish brew (allows it to be found)
Name brew
Brew characteristics calculations
Search for published brew
Database design
Register on website ability
Login
Inventory of created brews
printable format
Scaler

For this project the final product should of course contain all these features, but with these features in mind we can explore the functionality and desired behaviour of the system in greater detail. As such we can setup a series of use cases for desired system (The Use cases can be found in Appendix A).

A user of the system should of course be able to add the required ingredients for each recipe. As such foremost this is our first use case. As being able to use the recipes after having confirmed the calculations is an important feature, it should be possible to print them in a suitable format, at the same time the system should be capable of warning the user if the recipe flawed (such as missing yeast), as such we may model uses cases for these scenarios as well. Another core feature is to be able to save recipes such that the user does not need to renter them into the system every time he/she wants to make a change. Directly in line with this is should be possible to load the saved recipes. Logically allowing users to save data, requires them to have an account on the web site. The system should also allow for publishing recipes and thus requires a search function. This in turn means a suitable way of displaying the results of a search for recipes must be added, as well as a way of searching, for example either by name or brew type.

We should also be able to correctly handle custom ingredients, for example ingredients not already in the system. This gives the user more flexibility when it comes to ingredients not present in the system. In turn this means the application should be able to store the ingredient and that its attributes should be editable by the user.

Additionally users of the system should be able to add comments on others recipes and comments should be displayed when viewing the recipes. Finally it is worth while considering the life-cycle of items in the system, clearly not being able to delete ingredients etc. would eventually degrade performance significantly. As such it is important for the system to have some management tools, both to improve the maintainability of the system and to improve users experience. In this case especially, it might be worth creating a management system that allows for easy managing of default ingredients. These admin users might also be able to delete (or bann) other users and remove recipes that do not comply with site policy. General users should of course also be able to delete their own recipes and comments.

4.3 Vision

As stated in the project description the vision for the project is:

"To create a useful tool for recipe design for home brewers and microbreweries"

Why?: When creating a brew, it can be annoying and time consuming to change the proportions of ingredients to the rest, as this may also potentially result in inconsistent changes the recipe. This means its useful to have a tool that on the basis of this change, displays the resulting brew characteristics, without having to redo the hand calculations of the brew.
How?: The deliverable of the project is the web-site with the recipe calculator. The calculator in itself can be seen as useful tool, as it significantly eases the calculations on basis of a recipe.
What?: Making the web-site and relevant infrastructure is necessary to have a successful project, and as explored in the previous sections, we now have a some clear requirements for exactly what the system should be capable of doing.
Looking at the vision we can see a clear goal for the project, namely to create a useful tool. However the goal is not very easily measurable, as the view of a "useful tool" is up for interpretation. As mentioned earlier if one defines useful as for example, the system doing calculations for user, then by all means its easily measurable whether or not its a useful tool. The usefulness of the system could also be measured by whether or not it allows for the user to easily print the recipe, scale it and allow custom ingredients. As such to measure the success of the project, we must also look at whether or not the use cases are satisfied, as these are features a useful system is assumed to have. Furthermore we can also look at the stakeholder satisfaction with the resulting end-product, and use this as another success criteria. The primary stakeholder for the project being Christian D. Jensen, other stakeholders are the end-users, which could be the participants of the beer brewing courses at DTU or DTU brew-house. Their satisfaction with the system can thus be seen as another success criteria of the project.

4.4 Complexity

A good quick overview of the complexity of the project can be gained through the TQC model (also known as iron triangle). The limiting factor for the project is of course

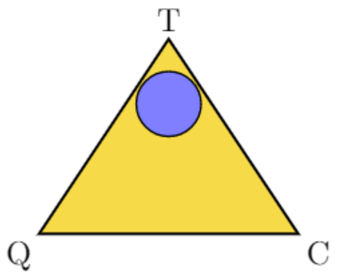


Figure 2: Visual representation of the component interaction

the time aspect, as the time frame of the project cannot be changed mainly because of "the turn in date" for the project. Furthermore with the limitation of manpower in the project one could argue that the "cost" of the project is already predetermined as well, which makes the only variable aspect the quality of the project. Under normal conditions this would mean perhaps less features could be implemented. However for this project the feature listing descried earlier has been made with a realistic assessment of time, and due to this the quality of the project becomes more in line with the requirements of the project description and the "cost" becomes the variable factor (for this project cost might be better assumed as man hours).

The scope of the project can usually be found in relation to both the vision and stakeholders. In this case the goal is to manage work such that all features can be completed in the time frame of the project and with proper quality. The project requirements (and features) are fortunately well defined in the project description, and as such it should be possible to avoid scope creep[3].

4.5 Uncertainty

When specifically looking for a risk assessment mythology, one could use the ideas and concepts presented by ISO 21500 and the DTU course "Project management" however this in itself does not focus fully enough on aspects specifically related to software engineering. In the case of this project we do not really concern ourselves with any supply lines, nor do we have any risks associated with multiple teams working together, it is clear we can focus our risks mitigation towards the software exclusively. It should be noted, that a security scope was set and agreed upon by stakeholder Christian Damsgaard, as the project is one of limited time and resources, the scope being the OWASP top 10 list[4].

[3]Scope creep (also called requirement creep, function creep and feature creep)

Starting with the two classical definitions of risk, specifically the risk of suffering

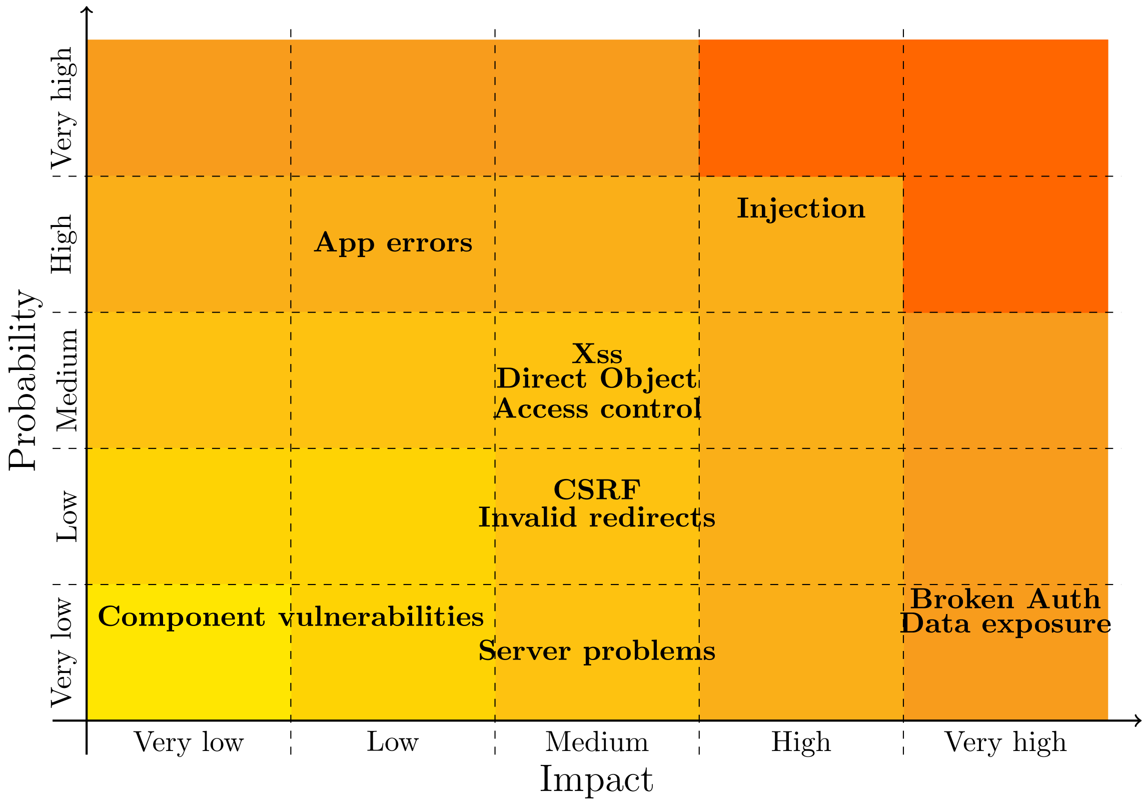


Figure 3: Probability impact matrix.

harm to the "company" or the inherent uncertainty in plans[5]. Firstly looking at the risks that may potentially harm the project/"company":

- Security issues as specified by the scope / owasp top 10.
- Service downtime / maintenance requirements.
- General service errors.

General errors in the service, are quite difficult to fully mitigate, though testing can reduce the likelihood of undesirable scenarios remaining undiscovered before release. Service downtime and maintenance can be seen as risk or uncertainty especially if the application becomes widely used internationally, as off peak hours may simply not exist.

In figure 3 we can see the risks sorted. One of the biggest threats is that of SQL injection, as such measures should be taken to avoid this problem in the first place. Xss, direct object references and missing function level control should also be avoided.

Via testing, we should try and reduce the possible application errors and minimise their impact. Given the severity of both data exposure and broke authentication these too should be avoided. In the case of this project it would of course be beneficial to reduce any undesirable risk, as such we can try to reduce the likelihood of these errors. Some parts of the risks associated with server problems however are up to the service provider, in many cases choosing a reliable service provider would greatly decrease the likelihood of this type of error and so we will accept it. The same can be said about component vulnerabilities, of course in this project it should be a given to strive for using components and libraries that contain no known errors. But we shall not concern ourselves with it any further. We can now gather this into figure: 4. This figure gives a good overview of how risks should be handled, some should clearly be dealt with directly, while others should be considered, but perhaps not as actively avoided.

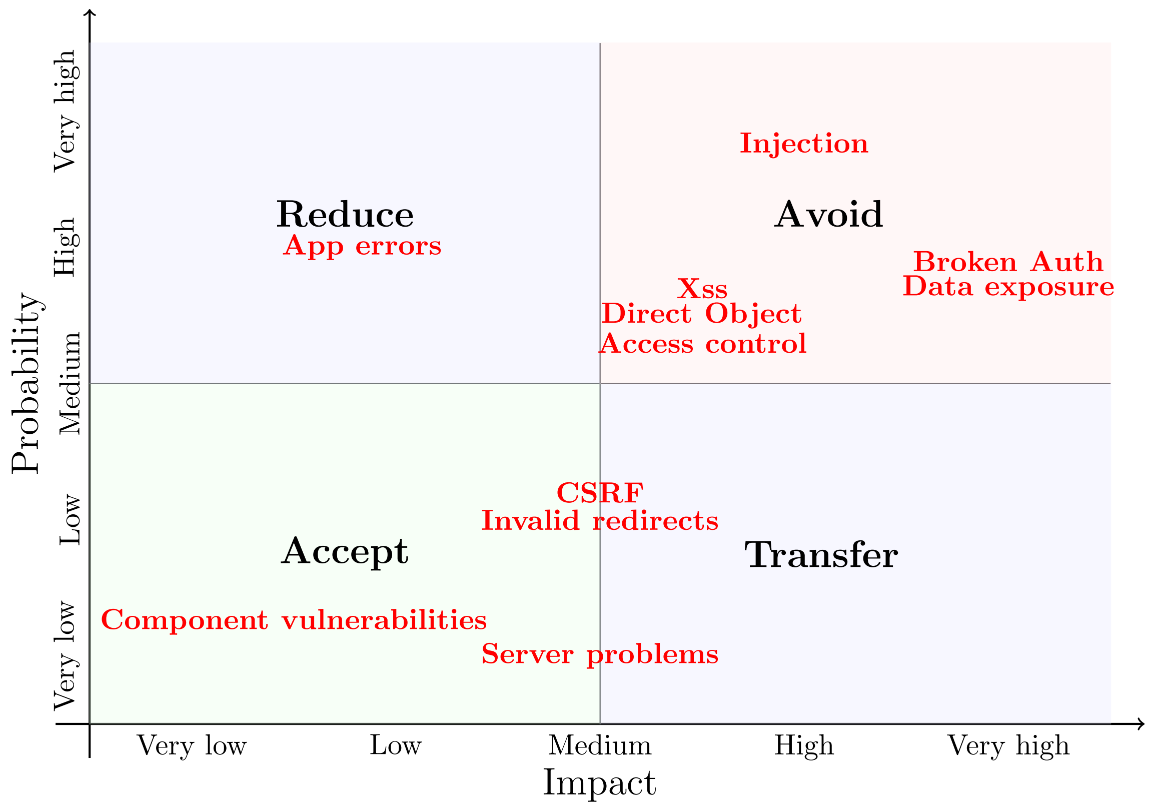


Figure 4: When to accept the risk in the probability impact matrix.

4.6 Database

From the use cases, requirements and feature listing we can quite easily make out several types of the ingredients the database must contain. The database must of course contain the different types of ingredients such as, fermentables, hops, yeasts, spice and mashsteps.

Table 1: Simple listing of database tables

User
BrewType
Fermentable
Hop
Yeast
Spice
Mashstep
Recipe
Comment

Additionally it should contain some sort of information, such as comments and users. In the case of creating a recipe we must bind it together somehow. Lastly we also have some information on top of this, namely somehow users should be linked to recipes and in turn comments should be linked to both users and recipes. Figure 5 shows a simple

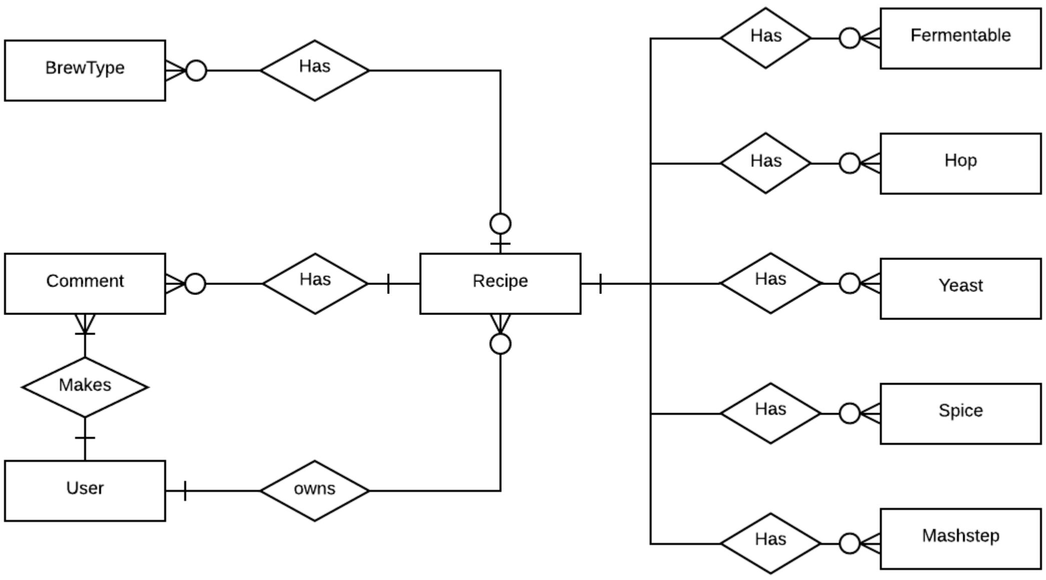


Figure 5: Simple ER representation of the database

Entity relationship between the tables [6]. Specifically a user can own many recipes, each recipe may in turn have several of each type of ingredient. A recipe may also satisfy one or no beer type. It may also have several or no comments, but each comment may only be linked to one user. In this case I have elected to not include information such as Attributes, as it could quickly become overwhelming for this type of diagram. Additionally the deeper one dives into the required attributes and requirements of the system it becomes clear that this simple structure of the database is insufficient for effectively storing default data types in addition to the data saved by the recipes. As such we must include additional tables to resolve this issue. Overall this can be seen as a naive data structure as it would only allow for comments on recipes, and not take into account different sorts of criteria such as a rating system, or comments different places around the site. Therefore this approach is not very flexible.

4.7 Application

Since we are designing a web service/site, some aspects of the user interface is predetermined. In this day and age it is pretty much well expected that there is some sort of navigation bar available to the user, such that he/she can easily navigate between pages. As such it is also logical to split functionality in pages of content, it also becomes logical that some of the pages requires authorised users only, such as a users collection of recipes. The diagram presented in figure 5, can be seen as centred around core functionality of the site. Our "recipe designer" ought to make use of all of these tables. During the design phase it is also well worth considering the life cycle of application items, such as recipes. A user might create a recipe, publish it, edit it and then delete it eventually. This does add some needed functionality to the application layer, but should ease testing somewhat, as objects can be created and deleted repeatedly.

As the application is web-based and core web technologies use URLs, it might be interesting to use this around the pages. Such that for example recipes are sharable via URL. This would also ease the user to user interaction, as sharing of recipes becomes very easy.

Lastly it might be worthwhile considering what a user of a modern web-site comes to expect in terms of functionality of such a system. It ought to be convenient for users to recover their passwords if lost and, it ought to be possible to for a user to change his/her passwords or email. In addition to this it should also be required that users have a place to easily find their saved items, such as a recipe and comments.

5 Design

5.1 Platform/architecture

While many people does not consider java when pondering the creation of a web-application, the matter of fact is that many large companies use java for this purpose. The reasoning for this could be that java is platform independent and thus runs on most computers, and that it could be considered a safe choice both in terms of hire potential (finding skilled coders of the language) but also in terms of documentation availability. On the other hand java does not allow for as fast as possible prototyping and release as other web-based applications (such as ruby/php).
Several java application servers exists, such as tomcat, glassfish, jboss or even ibms web-sphere among others. As the system is for homebrewers and microbreweries, the web service might be to expensive too host on a mainframe, moreover the horsepower of a mainframe would be overkill for this application. Specifically if the system requirements in terms of uptime was much stricter, websphere hosted on a mainframe would have been the best. This leaves the other options. Apache tomcat is a open-source web server, which implements both java servlets and java server pages. The server is capable of running on both Linux and Windows servers. Compared to the other java application servers, tomcat implements servlets and jsp specification, whereas the two others are full java EE servers (Java Platform, Enterprise Edition). Tomcat is however significantly less complex and uses less resources (eg. memory footprint). Furthermore it allows for modularity, such as MVC.
The model view controller architecture is a way of structuring the implementation of a application. As such we can also use this for the web application, as shown in the figure below. The model is responsible for storing and retrieving data, the view displays

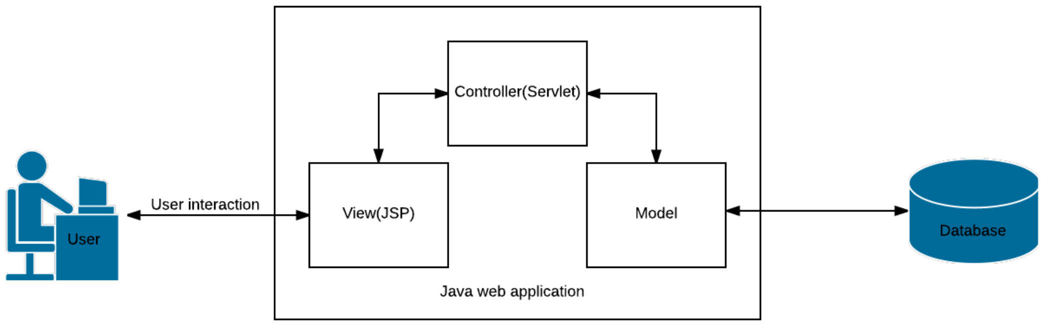


Figure 6: The model view controller architecture

the state of the model and the controller receives interpreters and validates input, and updates and the view depending on the need. In java the view is java server pages(Jsp for short) displaying information send by the servlet (controller) about the state of the model. The model in turn communicates with the database. Furthermore this architecture allows for easy testing of the model and the communication between the application and the database, testing the correctness of the functions. Additionally the model could be re-used for a mobile phone or desktop application if a different version of the software should be developed. As such it allows for some code re-usability and a way of separating the business layer from the presentation layer.

In the JSPs we can write code in HTML and add elements depending on input (from the database for example). This can be done with JSP scriptlets which can contain any number of java statements, declarations etc. variables themselves can be translated to text as well such that they can be added to the HTML code. This HTML code is what will be displayed in the browser, the java code embedded herein will not. This alone does not allow us to solve every scenario. Since the java code in the JSP is run on the server side it does not allow us to dynamically change the look of the page without sending messages back and fourth to the server. This is where javascript comes into use.

The key difference between the use of javascript and JSP is that JSP will never be seen by the user. That is to say it is executed on the server side only, all the client will receives, is the resulting HTML code(which could include some javascript). In turn the users browser may then run the javascript (client side). This of course means that critical code should not be run by javascript, but rather in the JSP. In some cases javascript, might not be enabled by the user, in which case this would leave the features programmed in javascript inoperational. However most modern browsers have javascript running by default, so it is less of a problem to assume that the user has not disabled it. In the case of this project the choice comes down to whether or not one prefers less communication between the server and the client. Javascript allows code to run locally on the client-side, so there is no need to send information back and forth whenever another ingredient has to be added or changed. In order to keep track of ingredients we would either have to keep the information live on the server or it would require us to send all related information back to the server every time a value is changed in the GUI. As otherwise it could mean the calculations in the UI are now wrong once a value is updated. Furthermore the repeated updates may end up causing users with less stable internet connections to lag slightly. In this case using javascript to add ingredients would allow for a more responsive UI.
While the communication might not be huge in terms of bytes, if multiple hundreds (or thousands) of users where to submit continuously every time an ingredient is added (a round trip has to be made) this would surely increase the required system resources and the cost of hosting the system (see the alternate solutions section in discussion). This is not to say everything can be handled in javascript, namely some things are better handled by the server side, such as saving data to the database, for example in the case of saving a recipe and allowing it for public display. Imagine the user names a brew something along the lines of:

$$" < script > alert("IAmAnAlertBox!"); < /script > " \tag{13}$$

In this example it adds a harmless alert box to be displayed, however this might be something completely different, such as a redirect or other actions that should not happen. We must therefore ensure that no illegal input has been added to the recipes that are to be published or saved (nor allowed to be send from the servlet to the database) and indeed whenever user input passes by the server side. We thus end up with figure 7.

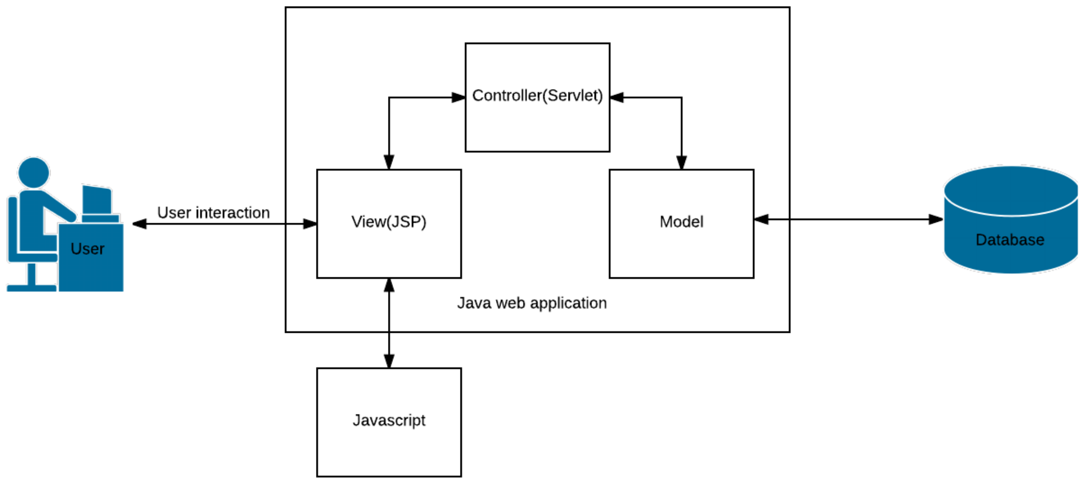


Figure 7: The model view controller with javascript

The javascript functionality manipulates only the state of the view, in turn information can be send from the view to the controller, which may update the model and through the model also the database. The controller can thus update the view with state changes from the model, which in turn stores data in our database.

5.2 Development method

In order to ensure a redline between the desired end-product and the use cases, it often becomes important to structure the development in a logical and orderly fashion. As I during the project am working alone, the methodology of extreme programming is impossible as pair-wise programming simply is not possible. Another method the waterfall model can often result in a disconnected end user product. Lastly Scrum development contains a large portion of planning for a single person project of this size. While the bachelors is a large project in scale of an education, it is not very large in the scale real life project and because of this the size of the planning aspects would take a considerable amount of time away from the project itself.

This leaves a slightly simpler option available, namely using a spin on the iterative and incremental development. While this is also an aspect of Scrum and extreme programming namely, that each iteration should result in a potentially releasable product, however in it self it is not enough to ensure that an end product is what the user needs.

Taking from Scrum the use cases to epics/features/backlogs concept and simplifying it

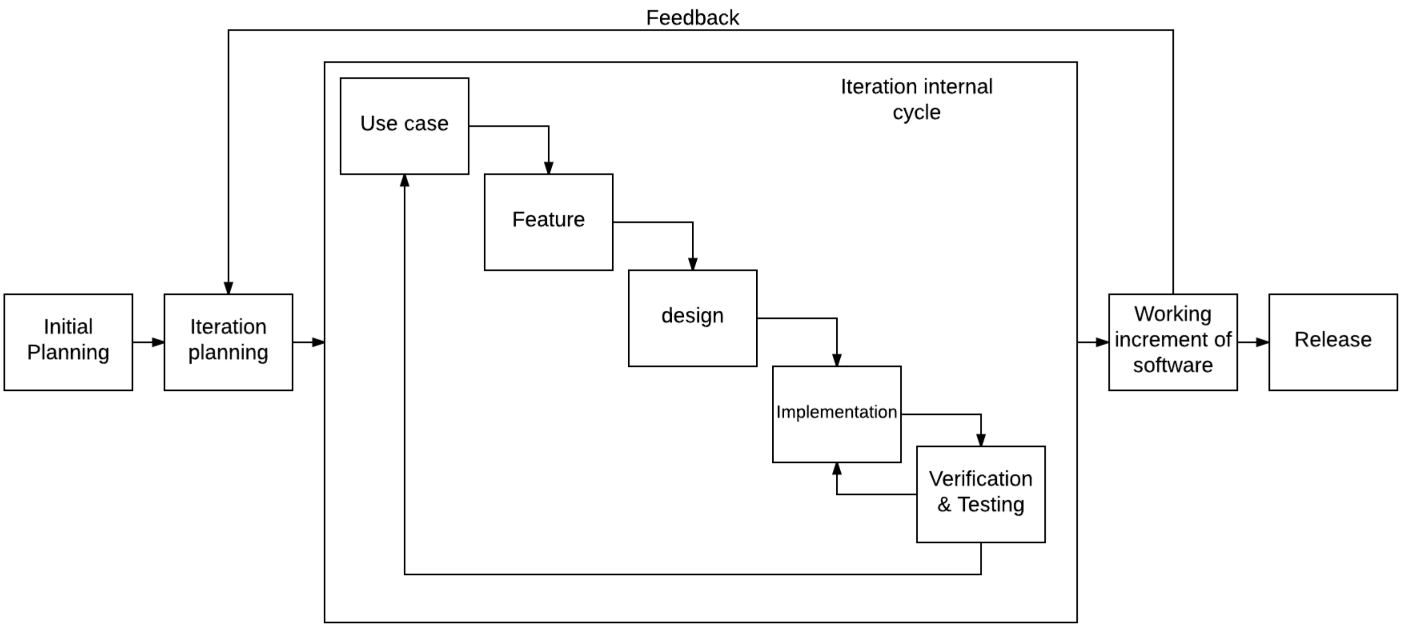


Figure 8: The combined development method

slightly would resolve this issue. For example we could create from the uses cases features and implement these. Furthermore creating some test cases on the basis of the functionality described in these and creating the model on the basis of these test cases would result in a good "red line" between the required functionality and the end user product. Furthermore splitting the project into potentially releasable products with the first being the minimal releasable product, ensure that even if a delay happens there remains a potentially usable product available to the customer. As such we can compile the idea together to form figure: 8.

The iteration cycle is derived from the waterfall method, however since each iteration is broken down into use cases and features and the waterfall methodology is used for each of these. In turn this idea somewhat closely resembles the less popular iterative waterfall methodology. There are however some notable differences, namely in terms of how the project is broken down into iterations. Here we break the project into uses cases and then features, create some test cases on the basis of these and then does the design and implementation, as such mixing aspects of Scrum and test driven development.

With the initial planning completed, one can split the desired features into blocks of functionality described by use cases, thus some planning for the block is required. For example this would be a good time to create extra use cases, and picking out which use cases and features are most important for this iteration. Once the iteration planning is complete one can start by picking out the most important use case and feature of that iteration, then doing a design and implementation phase and do verification/testing on the feature. One could further implement test cases in the use cases phase and add in some practises from test driven development, thus ensuring the implemented featured are also the desired functionality, strengthening the red line between requirements and implementation. This way of dealing with each feature during the iteration phase, ensures proper testing of the model. Once the verification is completed a new part of the use case (or a new use case all together) can be worked on.

As the blocks of functionality in this development model are not too large and the project itself split into subparts this should help avoid to large code recreations if a project changes during development. Furthermore it should ensure even if a substantial delay should occur, that at the very least there is a working increment of the software available to the client. Once an increment is completed a new iteration can be planed and executed. As mentioned before, a working increment may also be potentially released once completed. If following scrums increment size method, and splitting it into "sprints" of between 7-30[4] days, or it could be logically structured such that each increment is a desired bit of functionality. First increment could consist of the recipe calculator, increment two could consist of the login system / save / load / publish functionality and so on. It should be noticed however if taking from the increment size in scrum that, the initial sprint, (also known as sprint 0) is slightly larger than consecutive sprints. This would mean only sprint 0 and 1 are doable in the given project time, however a shorter sprint could be added.

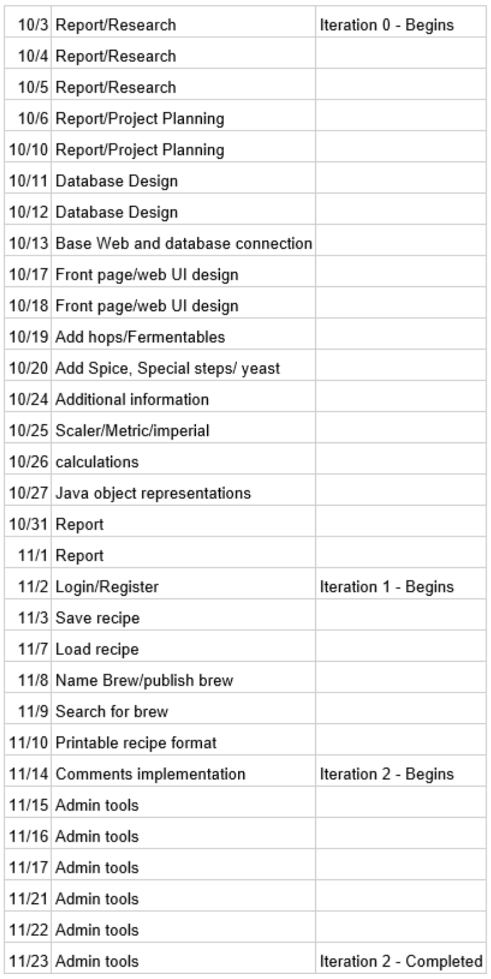
The major weakness of the proposed development model is that it does not split features into smaller units of functionality and this might result in poor management of resource allocation in a development team, as each features implementation size may vary greatly. In turn this could potentially create a critical path through the project that does not allow for good allocation of manpower within the project team. However this shouldn't be a problem for smaller teams.

Since this project is a one man project this pitfall is somewhat easily avoided as such there isn't many options for parallel work to happen in the first place, (Unless starting 2 features at the same time). As such it might be well suited for linear work in increments. During the results part of the project i will explore the impact of this development method and discuss its contribution or impediment to the success of the project.

[4]https://en.wikipedia.org/wiki/Scrum_(software_development)

5.3 Iteration planning

In the analysis section I described the desired features of the programme, now with the development method set and the initial planning done. We can plan for iteration 0. As the basis of the project rests on the database this is a logical place to start. Furthermore for a minimal product we require the functionality of being able to do the calculations for a recipe, as such adding hops, fermentables yeast... etc. and having the system perform the basic calculations described in the theory section. Furthermore each feature should also implement a basic UI, and some security aside from the application layer itself. As such picking out the most important features that would be required for an minimal potentially releasable product we would have something along the lines of figure 9:



10/3	Report/Research	Iteration 0 - Begins
10/4	Report/Research	
10/5	Report/Research	
10/6	Report/Project Planning	
10/10	Report/Project Planning	
10/11	Database Design	
10/12	Database Design	
10/13	Base Web and database connection	
10/17	Front page/web UI design	
10/18	Front page/web UI design	
10/19	Add hops/Fermentables	
10/20	Add Spice, Special steps/ yeast	
10/24	Additional information	
10/25	Scaler/Metric/imperial	
10/26	calculations	
10/27	Java object representations	
10/31	Report	
11/1	Report	
11/2	Login/Register	Iteration 1 - Begins
11/3	Save recipe	
11/7	Load recipe	
11/8	Name Brew/publish brew	
11/9	Search for brew	
11/10	Printable recipe format	
11/14	Comments implementation	Iteration 2 - Begins
11/15	Admin tools	
11/16	Admin tools	
11/17	Admin tools	
11/21	Admin tools	
11/22	Admin tools	
11/23	Admin tools	Iteration 2 - Completed

Figure 9: The work plan for iteration 0, 1 and 2.

Clearly the minimal product is a functioning website with the designer present. The next big leap, would be the addition of user based services, such as login, save, load etc.

Lastly (while not in the direct requirements), it is important for usability the system is maintainable, and therefore it could be important to consider some admin functionality, like adding default ingredients, admins being able to delete recipes, ingredients etc, this in turn is sprint 2. In the following sections we will not concern our selves with the sprint structure any further, but rather leave it to the result section to review the impact of iteration programming in this project.

5.4 Database

In the case of this project, using a database is pretty much necessary as the system exists online. Users expect fast replies and a databases can help us accomplish this. Had the system been purely a local desktop application or phone application, an acceptable alternative may have been plain text files. However in this case storing and searching through large amounts of data effectively might be done with a database. Although there exists alternatives to conventional databases(NOSQL), we will be using a classic database to accomplish this task. Here again there are many choices to choose from, however MySQL provides both a scalable and high performance database, and the default IDE presents some useful tools for the development. One of these, being the ability to create Enhanced entity–relationship (or EER for short) and from these generating a database instance containing the described tables, as such from now on we will look purely at EER diagrams instead of ER diagrams (an explainer for icons in EER can be found in appendix B1). As the database is the basis of the software and not actually directly part of the MVC architecture, it becomes only logical that it is designed before the model, such that there is something running that can be called more easily for testing.

5.4.1 Database

From the project description, use cases and analysis we know that the database must be capable of storing the state of a recipe as well as the ingredients them selves. Furthermore we know that there is functionality required for logging in and linking users to saved items. Lastly for the defined system we must be able to search for recipes and see relevant information regarding these saved recipes. To provide a basis for a dynamic front page, it is useful to have something in the database capable of storing some form of article or a simple news feed. For simplicity and to avoid scope creep, as this can be considered an extra functionality, we will assume that articles are not attempted published with the same name, on the same day. However stricter rules for the database could be implemented. Finally we also have the perhaps most easily expandable feature, namely commenting on others recipes. We can now start designing the database tables for the functionality described in the analysis.

Starting with the user table, especially here there is sensitive information, and the security of this data must be considered. Firstly the users should have unique names and emails. The user names ought to be unique as otherwise confusion as to who has made a recipe might occur, for example if two different users have the same name. Furthermore we should not allow users to create multiple accounts per email, as it would then be possible for a user to create literally millions of accounts for the same email. Here it would be possible to use the email as a primary key (and login credential) as well given the fact that its unique. However displaying peoples emails as a user name is not preferable due to confidentiality which may cause people not to use the site. On the other hand allowing multiple instances of the same user name might rightfully confuse users of the system. With a unique user name we also need each user to have a password as otherwise other users would need only know another name in order to login as them. This does present another issue, clearly storing passwords as plain text in a database would leave a large security hole in the system as anyone who should gain access to viewing the table would be able to directly decipher a user login. Therefore we should salt and hash the password, such that a potential attacker does not gain all the users passwords. In terms of encryption it might also be worth considering to add some layer of security to user information such as the email. However if we salted and hashed the email we would not be able to use it for anything other than comparing an incoming email to an existing one, as such we could not for example send emails to the users, since we cannot unhash/salt it easily. In this case it might be useful to encrypt the emails such that the encryption is reversible.

In the end to extend the system to allow for email recovery of accounts and in the future perhaps newsletters to be sent from our website we need the email and clearly storing it in plain text is worse than adding even basic encryption to the value. On the other hand this clearly is not enough to stop the determined attacker from deciphering it. If the system could avoid storing an email all together this would clearly be the best, as the safest data is data that does not exist. While this is better than nothing, it simply does not ensure an attacker having gained access to the database will not be able to gain the information he is looking for. Getting back to passwords, we never need to retrieve a plain text version of the password and as such we can salt and hash it as described in the theory section. Lastly the user table should also store our salt in the database so we can match incoming passwords with the hashed/salted ones, as such we must have an attribute for storing the salt. Using this basic encryption does require that a user does not have access to the system source files (Model for example) as here the basic encryption might be more easily be deciphered. As such we get the following 2 tables:

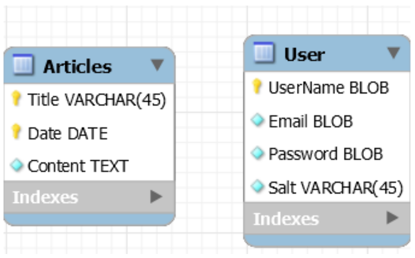


Figure 10: Enhanced entity–relationship diagram of articals and user

As mentioned in the analysis section, the first problem we have with the current sketch of the database is we have 2 types of stored ingredients, namely default and user created ones, additionally a recipes ingredient must contain information such as how much of the ingredient must be used, or in the case of the default ones which characteristics are related to them. Looking specifically at an ingredient, for example fermentables. We know a certain type of fermentable must have a name, it will add some sugar and some color to the batch. Therefore we must save this information inside the table. On the other hand default fermentables need not contain information such as weight and how they are used. As such we have two tables containing information, one for the fermentable and its attributes and one specified to the user input such as use and weight.

We now face two remaining problems, firstly that a fermentable must be able to tell which stored values are the default ingredients and which are recipe based, and secondly which keys to use as primaries. Yet again looking specifically at fermentables, we could use the name of the fermentable as as primary key, however this might present extra work for any application layer using the database. This being mainly caused by the application having to juggle the default ingredients if referenced by a recipe, if lets say, 10 users used the same default fermentable and one overwrites it, this would create a conflict with the ingredients used by other user as the name might not be changed. This can be somewhat easily solved by adding a unique ID key to each fermentable, this also provides a solution to out first problem. Using this ID we can define whether or not a key is in fact a default ingredient by maintaining a table storing which ID keys are default, while every key not mentioned in this table can be considered user created(or referenced). Furthermore this allows us to quite quickly determine which keys are to be considered default, as instead of looking through many records, we can look through the ones referenced by the default table.

As we see in figure 11 we define 3 tables, one containing the fermentable and its general

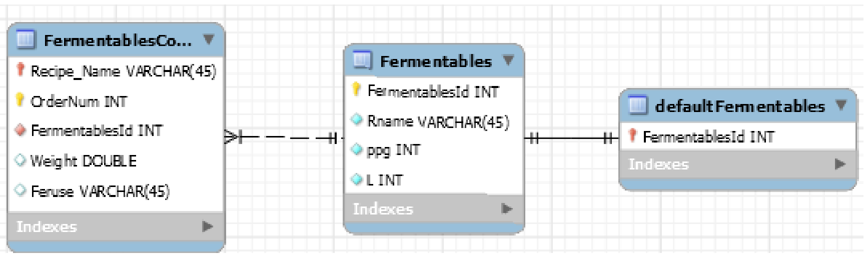


Figure 11: Enhanced entity–relationship diagram of Fermentables

attributes, a table defining the fermentables which are considered default ingredients and finally one containing recipe specific information. The one containing recipe specific information must be able to contain multiple instances of each ingredient, but be uniquely identifiable by the recipe they belong to. However here the recipe name alone is not enough to uniquely distinguish rows from one another and as such we can add a attribute describing in which order these ingredients where added to the recipe. This approach does assume the system requires unique recipe names, and for simplicity in this system we will assume this is the case. However this could as easily have been unique names per user.

We can now easily bind recipes together by the above presented approach. Each recipe will of course have its own attributes related uniquely to it, and as these are unrelated to one another, they can be contained within one table. Recipes themselves must contain a decent amount of information, since the system allows for both metric and Imperial measurement we must have this as an attribute. In this case what we are ensuring by storing whether or not a recipe is metric or Imperial, is that we save the true recipe. That is to say the recipe with no conversion errors(or rounding errors). Instead of just storing all recipes as one or the other. Additionally recipes have both batch sizes and boil sizes, as well as a total boil time and an efficiency, these too are of course important for accurately replicating a recipe. We must also consider which states recipes may be in, namely they can be publicly available (as described in the use cases) or they may be private to a user, the most simple representation of this state is a simple Boolean describing whether or not they are public. Finally a beer recipe, should have an owner, a brewtype and possibly a comment by the author. Recipes might not contain a brewtype and an owner in either case, a beer type might not be satisfied and in some cases a recipe might not have a user linked to it (for example if that user has been deleted). In case of users being removed it makes sense to keep his published recipes. With the currently presented approach we do however run into one problem, when looking for beers as a user, certain characteristics of a beer might be interesting such as the alcohol content for example. When searching it might be cumbersome for the system to have to fetch all a recipes ingredients and calculate the alcohol content for each recipe. As such it makes sense to maintain some sort of database instance of a recipes characteristics. Here it makes sense to include these in a separate table, the same goes for brewtypes, as these may be used elsewhere in the system.

Looking at figure[5] 12 we see the proposed solution to ingredients being modelled across the board. Additionally with the ideas presented in regard to brewtypes and quick search attributes. It also contains a simple solution to telling admins apart from general users. In this version of the system we only really have two roles (or types of) users namely admin and general user, where admin can be considered a "super role". A future version of the system could be upgraded to change the admin table to include other roles of users such as moderators for example, this would of course require a minor adjustment to the table admin as it would need to include a role type describing different types of users. BrewTypes are here implemented as min/max values, additionally we have no need to store calorie and alcohol content as these can quite easily be calculated from the other characteristics. Therefore they are not included in the table "brewAttribute".

[5]Full size figure can be found in appendix B

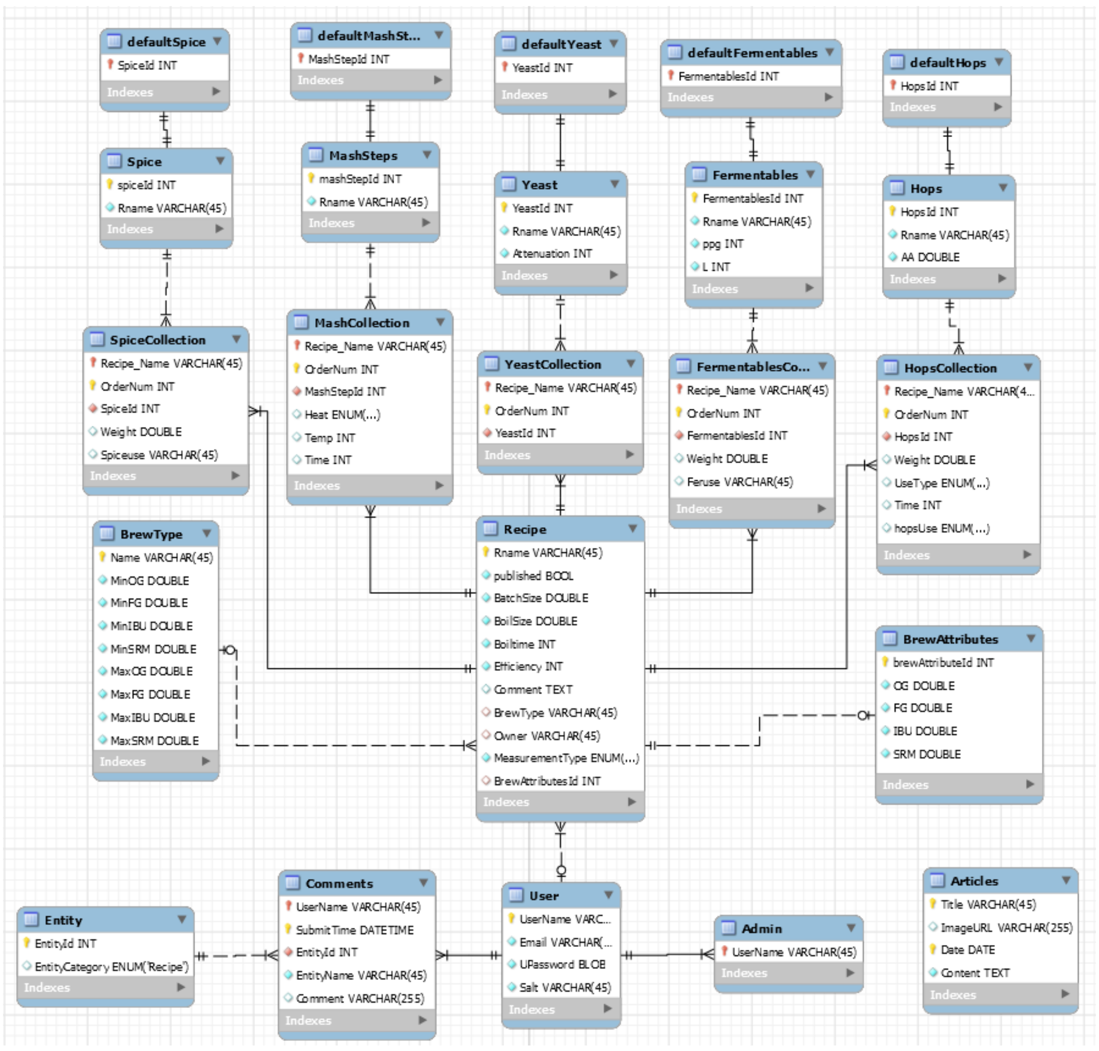


Figure 12: Full Enhanced entity–relationship without normalization

Finally the database contains the tables called Comments and Entity. The general idea of the table entity is to allow for easy expandability of the system. Namely the use of entity is to "point" towards other tables, as an expanded system might allow for comments on multiple things in the future. This could be comments on users, articles, and (in this case) recipes. However it is not unreasonable to assume a future iteration of the software might find use of comments being on articles as well. Additionally this approach might easily allow us to model "tags" and a "rating" system in the future as these to could point at entity and the entity table in turn at different tables.

5.4.2 Normalization

As stated in the course Database systems[7]: "Redundancy is the evil of all databases". Specifically data stored in more than one place, can quite easily become inconsistent, and inconsistent databases might in turn result in different perhaps even wrong answers to queries. The solution to this problem is to normalize and turn the database into a normalized version storing the same data. Some extra care does have to be applied when modifying the database for future updates, as these might not ensure the normal form is maintained. Starting with first normal form we have the definition of the normal form as:

- if and only if, in every legal value of that relation, every tuple contains exactly one value for each attribute.[7]

In this case we can clearly see that we fulfil the requirements of the first normal form as can be seen in figure 12, we can thus look towards the next normal forms. Looking at the definition for second normal form:

- if and only if, it is in 1NF and every non key attribute is irreducible dependent on the Primary Key.[7].

Here the normalization rules are a bit more complex, specifically if anyone table column depends only on one part of the concatenated key, then that table is not in second normal form. Once more figure 12 fulfils the requirements. We will now look at both third normal form and boyce-codd normal form.

- if and only if, it is in 2NF and every non key attribute is nontransitively dependent on the Primary Key[7]
- Non-binary tables (i.e. tables with 3 or more attributes in the Primary Key) is converted to several binary tables (i.e. the Primary Key consists of two attributes).[7]

Looking at third normal form, there might on the surface not appear to be anything keeping our tables from being in third normal form. However there is a transitive dependency in the table "BrewAttributes". Namely that if one was to change the attribute in for example Yeast (such as attenuation) clearly we would have inconsistent data present in the table "BrewAttributes", as the final gravity stored here would now be incorrect. Additionally, the problem may also apply to brewtype. Should for example the characteristics of a brew change, it might no longer fulfil the requirements of the brew type. As mentioned earlier the reasoning behind this is respectively that we wish to quite easily (without having to make a great many selects) be able to fetch information about the brew and display it to the user. In this case the whole recipe might not be important to a user browsing through recipes. BrewType on the other hand does not however directly maintain information as to whether or not a given recipe achieves the requirements it contains, but in this case is rather a tag, and we shall leave it up to the application to ensure recipes cannot save brew types they do not achieve. The brew type as such does thus fulfil the requirements of third normal form if one disregards the logic's of brewing.

Furthermore we shall also ignore the table "brewAttributes" containing transitive values, as this table can be consider a table used for less performance heavy searching (both in terms of number of database queries and application calculations). It does however add some complexity to the applications using the database as these must keep attributes up to date.

Looking at boyce-codd normal form, we can quite easily tell from figure 12 that our database does not contain more than 2 keys per primary key anywhere. As such we will say the database is in boyce-codd and third normal form (ignoring the before mentioned issues with third normal form). Further normal forms exist, however here we must also start making considerations as to whether or not, further normal forms are truly desirable. Fourth and fifth normal forms could be achieved but in this case would require further breaking down tables and would thus create more complex queries for the application layer to handle, (as we would need to insert data into more tables). Thus also decreasing the ease at which updates to both application and database can be made. Further additions of attributes might require bigger refactorings of tables and thus queries used in the application layer, thus reducing the maintainability and flexibility of the overall system.

5.5 Model

The model per the definition discussed in the architecture section, contains the communication between the application and the database, among maintaining a "state". A principle of web design is for the most part to design services as "stateless", of course in this case we must as a minimum have 2 states, namely signed in or not. Looking at the database communication, the system must be able to fetch and push information to the database.

An associated risk as identified earlier, was SQL injection, as such we must also deal with this during the creation of the database queries. More over we are interested in being able to store an object representation of a recipe in java. It might be smart to create object representations of most table objects, as to more easily transport data from the model to the view. As instead of sending 16 types of data, we can simply pass an object containing the same data.

5.6 Controller

The controller should fetch information from the view and incoming requests and generate appropriate responses. This being both in term of redirecting to pages, but also to ensure database calls are executed from the model in accord. Tomcat in the case of this application handles the sessions and as such can also maintain information such as which user is logged in. Afterwards it is up to the system to validate some aspects of it. Conventionally it might only be possible to have one controller (servlet files), however in this case there may be merit for splitting the controller into multiple controllers, specifically for different types of functionality, (front page, user and admin). Some care will be required in order to ensure these work together, however it should allow the code to be more easily navigable. Furthermore it might provide a good split between general functionality and admin functionality. Any one request may however only access one of these controllers, thus keeping with the MVC architecture.

5.7 View

The choice of creating a web application does require some flow considerations, namely as mentioned before most states should be reachable from any other state. Mainly due to the navigation bar one comes to expect as part of a website. As such it is also worthwhile to ensure requests are not dependant on a state, but can rather be called from anywhere and is thus stateless. However it is still worthwhile to consider the flow of the application as a whole. Looking at figure 13 we see an idealised scenario for the flow of the application. As this is a website most states should be reachable from any other state, as the main pages consist of the "rounded" menu options these should of course be reachable from any page. Additionally the register and login forms should be reachable from all states as well (assuming the user isn't logged in already).
This is of course an idealised situation, as it is an expectation for a website that you do not need to firstly enter the front page, to reach other pages. For example a user might use a link provided by another user, and enter the designer directly, not mentioning the many states that do not require logins directly. Furthermore it seems advantageous to design the services such that they can be reached regardless of state (with some exceptions). This idea would also allow for better backtracking in the application as it would allow users to use the "go back" functionality included in most (if not all) modern browsers, which would make the page greatly more usable. Finally it is also worthwhile considering that some pages should have an easily linkable appearance, e.g. when a URL is requested it results in for example, the same search or recipe, (assuming of course a user or admin does not remove the given recipe) and as such can be seen as idempotent.

- http://localhost:8080/BrewRecipeSystem/designer?account=Dtu+beercalc+pilsner+1

For example if a user wishes to share the above link with another user, it is clearly advantageous that when the link is clicked, the user is redirected to the specified beer. Here we might also rightfully consider a scenario where the recipe has been removed, and the application should be able to deal with such a scenario by redirecting to an error screen, or an http 400 error code page (404 not found). This concept of web design also allows the site to be significantly more easily sharable.

It might be noted by looking at figure 13 that it does not concern itself with the flow for administrator users, however this can be designed as an addition to the figure, namely as the menu points under "admin page", and have functionality such as managing ingredients, users and articles.

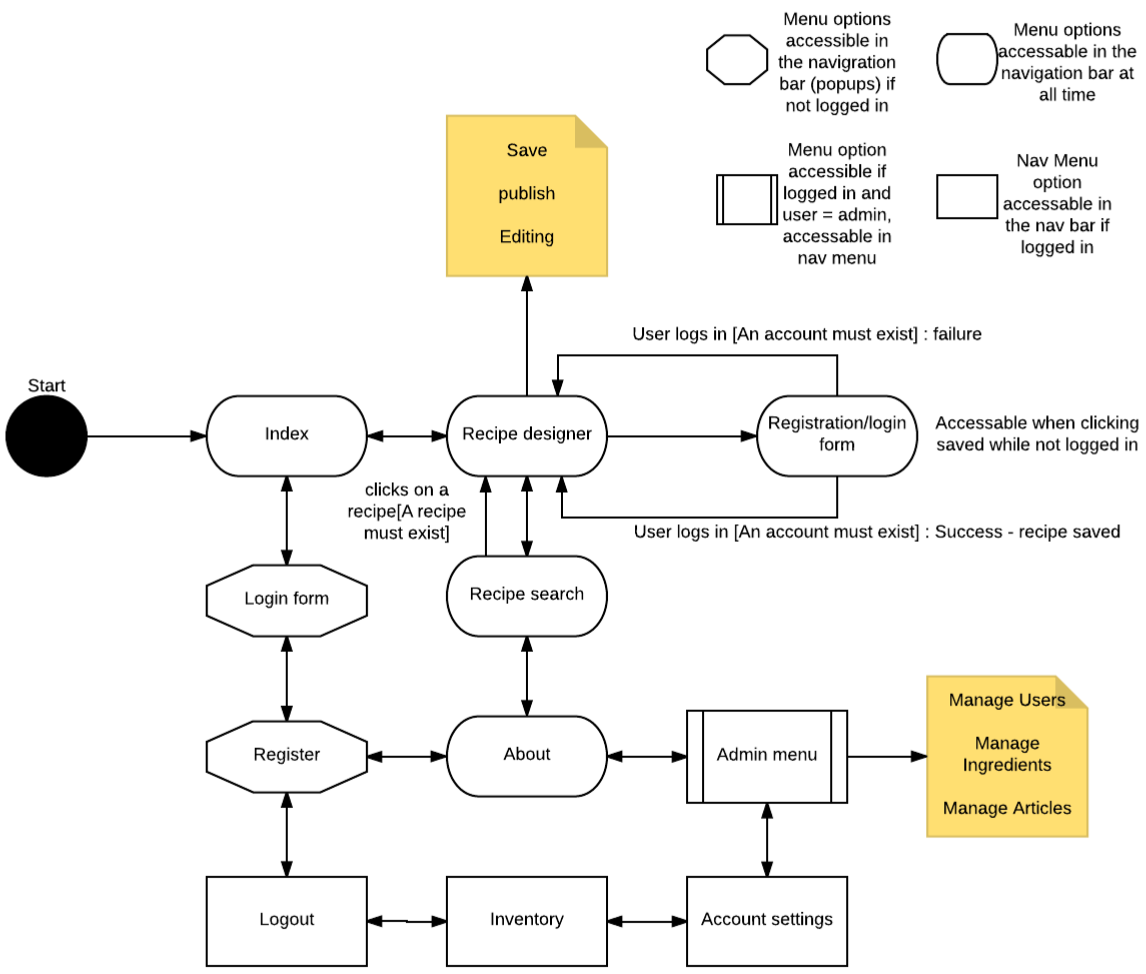

Menu options accessible in the navigration bar (popups) if not logged in
Menu options accessable in the navigation bar at all time
Menu option accessible if logged in and user = admin, accessable in nav menu
Nav Menu option accessable in the nav bar if logged in
Save
publish
Editing
User logs in [An account must exist] : failure
Start
Index
Recipe designer
Registration/login form
Accessable when clicking saved while not logged in
clicks on a recipe[A recipe must exist]
User logs in [An account must exist] : Success - recipe saved
Login form
Recipe search
Manage Users
Manage Ingredients
Manage Articles
Register
About
Admin menu
Logout
Inventory
Account settings

Figure 13: "Statemachine" showing the flow of the user interface

6 Implementation

6.1 Database

Specifically there are 2 SQL implementations of importance, one to create the database, and one to populate it with default data. One of the tools of the MYSQL workbench is that it allows for forward engineering of the database from the EER diagram (known in the workbench as Models). This automatically generates the code needed for creating the tables of the modelled MYSQL database. Here we have both the options of automatically deploying it, or creating a runnable deployment file for the database. A simple population script can also be created in order to add some custom (or default) data to the database, such that at any point the database could be fully reset. Additionally some functionality might justly be added to the database side, such as events like backups and automated checking for no longer used data. The system for example should allow users to leave the site, this however does not mean we should delete their public recipes, however the system would no longer have a use for their unpublished recipes. This means we might in the future rightfully have an event scheduled to clean up such recipes and other data that might not be used. This does present some challenges as this event could be very heavy on the database performance wise, and as such we would need to place it either off peak hours of the system, or somehow limit it slightly such that the system as a whole remains responsive.

6.2 Model

As we discussed earlier in the design section, it is quite useful to maintain object representations of database tables. As such we can implement java objects representing our database tables, although we have a need for less objects. Namely we need one for each type of ingredient, the recipe, brewtypes and comments. These objects contain much the same data as the database does. However here the object Recipe contains arrays consisting off the ingredients, the recipe uses. Arrays is also a convenient way to store the lists of default ingredients. Lastly its also convenient to have some objects representing search data, specifically to give users of the system a better overview of search data, such as the name of the beer, its type and some of its characteristics (such as alcohol bitterness, calories). In this case it seems these are the most important characteristics, as recipe based characteristics can be changed using the scaller.

Communication in the model with the database happens through embedded SQl statements, these have a somewhat repeating pattern, notably of the format:

Listing 1 *General structure of database queries in java*

```
PreparedStatement recipeStatement = null;
try {
    recipeStatement = conn.prepareStatement(/*Statement*/);
        //Prepared statement data
        recipeStatement.execute();
        conn.commit();
} catch (SQLException e) {
    //SQL error
}finally {
    try {recipeStatement.close();}
    catch (Exception e) {/* ignored */}
}
```

The above code snippet is a generalised example of embedded SQL in java, here the use of prepared statements is also a safety feature, as SQL injections will trigger the SQLException catch block. SQL queries are simply inserted instead of the statement comment, any "?" in the SQL code can be filled in by prepared statement commands, and additionally the whole SQL query is executed, at this point depending on the statement it might also be possible to fetch results from the query. Finally we ensure that the resources used (preparedstatement, resultSet) is correctly closed as to avoid a resource leak. Here it is worthwhile to allow multiple java methods and in turn SQL queries to be run via one connection and as such it should be possible to close the connection at an appropriate time, as otherwise this too could create a resource leak. In this case to avoid frequent reconnects to the database, it has been made so that each model class using connections, has a method specifically for closing that connection.

6.3 Controller

Requests or rather input from the user interface (view) is processed by the servlets, depending on what events occurred, different functionality is run, such as redirects, fetching and forwarding of data from the model and view respectively. Validation of input is also done here, as data from the view might be unsafe. As such we must validate the data send by the view. To avoid cross site scripting it is a good idea to limit the size of incoming input, although this is not always possible in systems, as some input may be extra-long, such as comments in recipes. A general principle in security is that: new is bad and homemade is often worse. Therefore it becomes necessary to use javas extensive free libraries to help sanitise input data. In the case of this project I use a library called JSOUP. JSOUP provides more functionality than input validation, for example, there may be situations where some HTML is allowed in input and JSOUP allows for filters to be used to specify the level of HTML and script allowed in input.

The behind the scenes functionality of the servlets mean that each time a server receives a request for a servlet the server spawns a new thread and call service. The service in turn checks the http request, and directs it to that method. While more http methods exist, I use only doGet and doPost. A doGet request results from a normal URL call or specified by the HTML form, while doPost is called specifically if a form lists the request as post. In the case of the doGet requests, we can guarantee calls with URLs return, the same results (assuming a recipe is not removed etc) and as such links to for example a recipe always result in fetching that recipe. DoPost on the other hand depends more full heatedly on the form input.

6.4 View

The view is made up of 3 main components, JSP files, CSS layout files and javascript. The JSP serve the need for dynamic content upon loading a page and is run server side. HTML components can be added in an embedded fashion, such that in the JSP file there is for example a java for loop which creates HTML selection options. As only the HTML parts of the JSPs are send to the client, it becomes important to have the java functions of the JSPs display the information send by the controller in an HTML format readable to the user. Additionally CSS provides overall structure and functionality for some components, especially the help tips are relying largely on CSS to function correctly. The "events" of the view can be described as forms capable of being submitted with a series of attributes, within the scope of the $\langle Form \rangle$ tags. The forms specify a servlet and method that should be called, and passes relevant information on to the servlet (controller), the servlet then executes functions depending on which form/button and input is available to it.

When talking about the view there is one more interesting feature, namely that the JSP files are not directly accessible by URL, the pages must be accessed through the servlets (except the defaultErrorPage, which is directly accessible to users and tomcat). This can also be seen as a security feature as, for example the admin JSPs are not directly accessible (and as such displayable). This means a user will not be able to access admin functionality without the server side allowing it. It of course still remains important for the servlets to validate and ensure a user is an admin before allowing users with access to these pages to do anything with them.

6.5 Javascript

Javascript is used to make the designer reactive, as such a large portion of its functionality is run on the client side rather then passing information back and fourth between server and client. The design of this functionality is pretty straight forward, it simply pulls information directly from the designer view and updates it. In turn it completes some calculations in regard to brews characteristics and thus also adds to the view, namely in the designer the user has the option to add more ingredients, set targets, calculate scaling etc. All this functionality is run by javascript and adds additional content to the user interface of the designer, while most of the validation of recipe, saving and loading is handled on the server side. Javascript also controls the pop ups around the site, specifically, register, login and account information forms. We will in the discussion section look at and compare round trip communication with the client-javascript solution.

6.6 General implementations

Generally most functionality at a minimum have the options to create and delete items. Recipes and articles are however editable and re-savable. Most admin functions, and comments do not allow for editing once created, but instead assumes it acceptable to delete the item in question and make a new one. This however does not provide problems for recipes referencing these default ingredients, as they remain in their respective recipes, now as custom ingredients. It does however leave a minor problem for brewtypes, as these are directly linked with recipes, as such when a brewtype is deleted so is its references in recipes, however as the recipe is otherwise left unaltered, this presents only a minor issue. Alternatively, we could solve this by not allowing brewtypes referenced to be deleted, however this approach presents its own problems[6].

Incoming login, and registration requests are handled through the servlets, here some

[6]Allowing editing of brewtypes might solve both issues. To avoid feature creep however, we shall however allow types to be deleted.

checks to the validity of data is run. In terms of registration it is checked that passwords are of proper strength, format and length. Similarly user names are processed. A new salt is then generated by the system, and the password salted and hashed using the "PBKDF2WithHmacSHA1" algorithm. Unlike other encryption algorithms PBKDF2WithHmacSHA1 is considered one of the safer options, that have yet to have security flaws discovered within it [7]. Here emails are not validated[8], but encrypted via the AES (acronym of Advanced Encryption Standard). The validated and encrypted data is then send to the database long with the salt and user names. If successful in creating a user, he/she will be automatically signed in, otherwise an error message is displayed.

Saving consists of submitting data consisting of all input fields displayed in the designer as a form. Once more the controller validates all input and translates it into an object representation of the recipe. The recipe is additionally validated and if successfully so, saved/or resaved to the database. Resaving requires some extra care, as in case of extra ingredients being added to a recipe, these must be added, and otherwise in case of the recipe containing less ingredients, these must be deleted. Additionally extra care has been taken into ensuring maintenance of the characteristics table (also known as "brewAttributes"), such that recipes are easily identifiable and kept up to date. If a recipe is requested and loaded, the recipe is first fetched from the database, turned into an object representation and forwarded to the view, which displays it as HTML on the client-side. Comments are very simply implemented in the application layer, as comments can be both added and deleted. As with all input comments go through the same validation to avoid Xss, once validated a query is run to add the comment to the database. The user can then if needed delete his/her comments from the inventory. The search functionality is created using MYSQL search functions utilising both MYSQL wildcard searching and in addition limitations on result sizes, as to improve load performance. The functionality of searching is also handled by the "doGet" methodology, as such searches on the site are repeatable via URL. When picking a search result, the doGet http request is run such that the recipe is sharable.

The admin tools are created such that some are directly usable from the respective pages and some through an admin menu, both visible only to admin users. Clearly we can implement admin deletion directly into the search function, as to reuse our search page. On the other hand the ability to delete and create new ingredients, brewtypes, articles and deleting users is clearly best left to a separate menu with separate pages. In larger projects it quite often becomes convenient to use the languages native libraries. In the case of this project we are required to use at least one package, in particular the package that allows for MYSQL to be run embedded in the java code. MYSQL provides this library to accompany and connect to their database, though it could be

[7]`https://www.owasp.org/index.php/Hashing_Java`

[8]We could however extend our mail system to send users a code for email verification, which must be entered before an account is considered valid.

used to connect to other types of databases. During the project, the validation of input is especially important and thus rightfully it is useful to use a well tested solution to counter Xss rather then creating one from scratch, in this case the library in question is called "JSoup". Lastly in the project we make use of a PDF creation library, here the choice came down to two libraries, on one hand a very easy to use and implement tool "itext" and the more low level library PDFbox, made by Apache. While in almost every regard the itext library was more comprehensive and easy to use, its licence presented certain difficult to judge legal issues for the project, and as such Pdfbox seemed like the safer choice, in order to avoid legal issues for any deployment of the project.[9] In addition use of googles mail service may be used to send mails via a gmail account to users.

We may now look at some of the more difficult segments of code to explore their implementation further.
Firstly we will look at saving. In the Controller.java file exists the dispatch code for the operation. If a call of the type save is received, the controller will call one of two methods, namely rs.update or rs.commit. If and only if a recipe exists and belongs to the current user, the application will run the database queries for updating, otherwise the commit. Looking at commit we might first have to consider one important aspect, namely the case where two users try to commit a recipe at the same time with the same name. As such we must add some point, to which only one user can proceed and is allowed to save his recipe. We create the items for the characteristics table, the primary key is then passed to the recipe creation which generates the recipe, if and only if after the creation of the recipe the recipe belongs to that user, is the rest of the recipe saved. Hereafter the appropriate data in the recipe is saved using prepared statements. Now looking at the update, there are 3 primary scenarios, namely if user is saving less respective ingredients, more, or the same number. If it is the same number we may simply overwrite the existing ingredients through SQL update commands. If more ingredients exist, we update the existing and generate any excess's ingredients similarly to our commit. Additionally if less ingredients have been used we create delete statements for these and simply remove them upon the update. Both methods also validate the correctness of brewtype to ensure a brew not satisfying these are not labelled as such.

We may describe our desired concurrency between two users, by modelling the operations using a petri net [8]. Looking at figure 14, we see the petri net, however one might notice in this representation, only one process gets to proceed, whereas the other cannot continue. In the case of the code implementation, it will simply quit the operation, whereas the other is allowed to proceed.

Logins logic persists in the controller under the method "login", in this case starting with a basic counter to brute force attacks limiting the login attempts[10]. Should the

[9] It should be noted itext has a free to use licence, so long as the source code is freely available to the public. However some criteria of the licence where hard to judge the consequences of.
[10] An alternate more advanced solution to this would be using CAPTCHA.

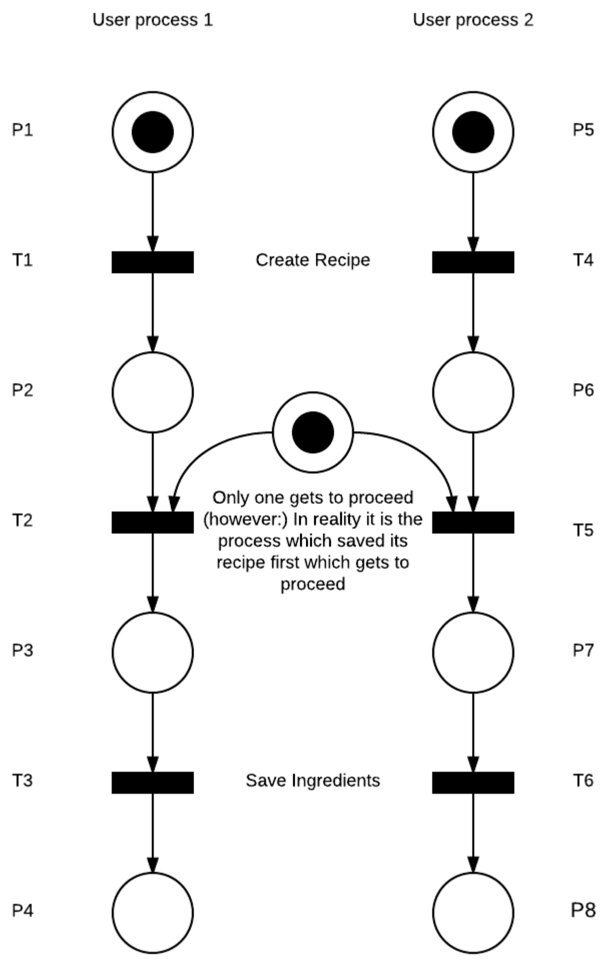

User process 1
User process 2
P1
P5
T1
Create Recipe
T4
P2
P6
Only one gets to proceed (however:) In reality it is the process which saved its recipe first which gets to proceed
T2
T5
P3
P7
T3
Save Ingredients
T6
P4
P8

Figure 14: A Simple petri net describing the desired concurrency

login attempts not be exceeded, the system proceeds to matching incoming passwords to existing ones, by pulling information (if it exists) from the database, if successful, the session is invalidated and assigned a new session id. Finally it is asserted whether or not a user is admin. If login is unsuccessful the number of login attempts increases.

As discussed earlier, we have two types of encryption, one using AES to make reversible encryption for emails and one for salting and hashing passwords. All encryption functionality has a separate class file, namely "EncryptionService.java". Firstly the class contains a method for generating random salts, which are generated through the SecureRandom instance to ensure salts do not become predictable. Following the salt generation is a method for hashing, which takes the password and salts/hashes it, the hashing happens a given amount of times using the "PBKDF2WithHmacSHA1" algorithm given by java. The class additionally contains encrypt and decrypt methods for

AES, and a password generation method for recovery of lost passwords[11]. Lastly the method contains a few support methods such as toHex.

6.7 Tests

Since some input especially names are specified to be of a minimal size, we can use shorter names for test cases, for example names of users must have a minimal length of 6 characters and recipe names a minimum of 4. As such we can use smaller names in the automated test cases to ensure they do not already exist in the database, additionally the delete functionality of the application allows us to reset after test cases. However this is not applicable in all cases. In order to ease the repeatability of the unit tests, the "@after" block can be added to the test code, the point of this special test method is to run after each normal test method. This allows us to create a "reset" script of sorts embedded into the test code. The point of the script is of course to reset/remove test data from the database, such that it can be rerun.

6.8 Security

As identified by the risk analysis the problem of SQL injection ought to be countered or at the very least reduced significantly. To this effect the use of the prepared statements have been used and catching any possible exception thrown by these. Additionally in order to avoid the most common forms of xss, all input from the GUI (or client side) is sanitised. This approach also avoids some forms of insecure direct object references, however in order to avoid others the use of doGet methods is quite limited and all account information and login information is send via doPost. Direct access to pages have also been restricted such that almost all pages can only be accessed through servlets. Some pages are additionally only accessible to users of a certain privilege (i.e. logged in /admin). Furthermore in order to avoid broken authentication and session management, once logged in session ids are invalided and granted a new session id, and once logged out, old session ids are invalided as well, furthermore session ids are not being exposed in URLs and do suffer from time-outs in case of user inactivity.

[11]Password recoveries, simply generate a random password and sends the new password to the users email.

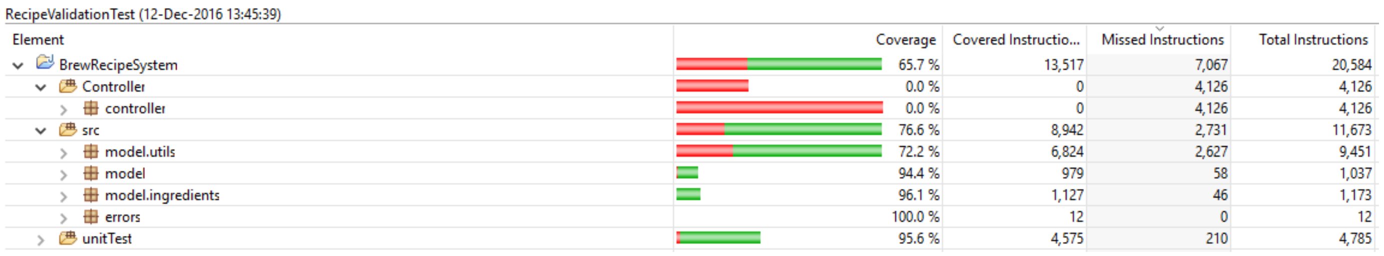
7 Results

7.1 Use cases

In the project, use cases have been used for multiple things. They have been used as descriptions of desired functionality and thus also integrated into the development method. They have been used for automated testing and functional web testing. The use cases where created directly from system requirements, giving ideals of behaviour for the system and thus directly influence the design. The completion of the use cases can also be seen as a release (or success) criteria of sorts, namely the successful implementation of these. As we will explore further in the following sections, the use cases have been completed along with their alternate scenarios. These having been tested both by hand and by JUnit tests. Given the use cases describe useful tools and functionality for the system it can be said that the vision has been accomplished as well, it should however be remembered that this alone is not enough to deem the project a success, but rather a good step on the way.

7.2 Unit tests

In order to secure the program against inconvenient errors it is very important to make some systematic software tests. As identified during the project analysis it is worthwhile to try reduce the likelihood of application errors and unforeseen behaviour. Here systematic tests may also help provide easy integrity checks for code once changes are made to the code. The application layer, consisting of a great many prepared statements methods which contains difficult to test code. Specifically SQL errors are difficult to test for, yet not impossible. The SQL exceptions can be triggered by several situations, such as (but not limited to) input exceeding database specified limitations (input length), or by illegal input, and by inputting duplicate primary keys. While test cases could have been added for all SQL exceptions I decided to spend more time refactoring and polishing some features identified by user input. When working with systematic tests,


RecipeValidationTest (12-Dec-2016 13:45:39)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
BrewRecipeSystem	65.7 %	13,517	7,067	20,584
Controller	0.0 %	0	4,126	4,126
controller	0.0 %	0	4,126	4,126
src	76.6 %	8,942	2,731	11,673
model.utils	72.2 %	6,824	2,627	9,451
model	94.4 %	979	58	1,037
model.ingredients	96.1 %	1,127	46	1,173
errors	100.0 %	12	0	12
unitTest	95.6 %	4,575	210	4,785

Figure 15: The code coverage by automated test cases

it is worthwhile remembering tests should not be written just for code coverage, but rather to test functionality and behaviour. With this in mind and looking at figure 15 we see the overall test coverage of the model to be 76%. One may note the "utils" folder having a significantly lower coverage than other packages, this being due to the bulk of database connectivity and SQL exceptions being present within the classes of that package. Additionally the coverage was reduced further by the use of enums, which unfortunately has some sub classes which are difficult to test completely. These do however not account for a very large % of the missing coverage as this is mostly caused by the SQL exceptions. Finally, the use cases have been implemented as test methods testing the correctness of embedded database queries, encryption and mailing services among others (see Appendix A for use cases).

7.3 Web testing

While it is possible to create automated test cases for the controller it requires quite a lot more effort or the use of additional manpower for personal testing. Here the testing of use cases provide some guideline for functional behaviour which can be tested by hand. Additionally it is worthwhile testing how some concurrency issues might effect the system, such as deletion of default ingredients being used by users concurrently creating recipes. Obviously it also becomes important that functionality can be used in a random order, i.e. the user is not bound to do things in a specific order. As such a smaller beta was held in order to ensure the correctness of operations. No concurrency issues where identified during this beta test, however some places where tooltips might be useful was identified and improved upon, specifically tooltips where added to the registration and login pages. Lastly the beta also identified, that users may input very large amounts of ingredients and thus break the displayed numbers somewhat. It should be noted however that the system does not accept saving such broken recipes and displays these problems as error output, additionally this is not necessarily a problem with displayed characteristics as the results are correct, however it is doubtful at best, that a user would succeed in doubling the density of water in real life without notable side effects.

7.4 Security/redundancy

The testing of security has been done through manual testing and off the guidelines that can be found through OWASP's top 10 project. Security while undoubtedly an important aspect of every day systems, is an extra none-functional requirement for the project. In this case limited by the predefined scope, and as such while providing a good security base could be well expanded upon for future updates.

Redundancy in the database has also been sought to be reduced by normalization. Especially attributes that can be derived from others have been identified and minimised (with the exception of search helper data). Minimally usable primary keys, and the elimination of repeating groups in individual tables. While higher normal forms might further reduce redundancy, it should be noted that there have been found a good mix between usability and normalization.

7.5 Development method

In projects in general it is often beneficial to look back at the process and carefully analyse what worked and what did not. In this way when moving from one project to another or indeed from one phase of a project to the next, it becomes a learning experience. As such it also becomes worthwhile to look back at the development method and review which points provided good structure and at which points it became a hindrance. As noted in previous sections (or rather predicted), the development method provided a good line between requirements, use cases and implementation, as these are logically connected. Additionally the project provided a good work structure for very linear or rather one man project. However ever so often functionality and database functions where split well enough that multiple people could have easily accomplished tasks concurrently.

With the database and java objects defined, functionality as according to the work plan became very split. Perhaps because of the parallels or inspiration of Scrum, the development methodology turned out to appear capable of suiting smaller teams working on a project. Because of the built in feedback mechanics in the method, work load for each task was better judged as the project progressed, leaving room for additional desired stakeholder features to be added (admin management tools). Due to the initial over estimation of task complexity, iterations where completed ahead of time and an additional sprint could be planned and executed, thus providing a more maintainable system overall. The direct link between use cases and functionality also provided a good structure for use case to featured based implementations. An additional side effect of the combination of interactive, and use-case to feature developments, is that in turn the split of functionality, caused more working increments of the software than first expected. Often upon completing verification of a given use case in sprints, the software could be considered a working increment. This perhaps being due to concurrent implementations of features not being possible (in terms of manpower).

8 Discussion

8.1 State of the project

The web application in its current form, fulfils the use cases defined at the start of the project fully. In turn it allows for recipes to be created, saved, shared and for users to comment on each others work. Additionally the system allows for printing of recipes, and finding recipes others have published. In addition to this, the system has been well tested for the correctness of functionality, and the database been designed to be functional as well as with minimal redundancy.

While not directly a requirement as according to the problem definition, some security and management tools have been implemented with thought of increasing the usability and ease the maintenance.

It would of course have been interesting to continue improving the application security, however overall in a project with limited manpower and time, and with an agreed upon security scope, this seems well in-line with requirements. If further work was to be carried out on the application, added security and additional refactoring would be a priority. A good starting point, may be to implement use of SSL (as well as https) and moving passwords, used statically in the code into a password file. In addition to this it would also be possible to increase the number of automated test cases overall, perhaps creating test cases for many of the SQL exception blocks.

Additionally there would be good point in creating some clean up events on the database side, specifically, when a user is deleted, his/her recipes are not. A side effect being unpublished recipes belonging a given user become unreachable from the UI. Therefore certain scenarios might leave footprints in the database and it would be possible to create a scheduled series of events to clean these up.

Further work on the UI might also be a priority, specifically improvements to the overall layout and site design. In addition to this it might also be worthwhile to phase out the use of scriptlets and replace them with javascript or JSTL. While this would not directly influence performance or functionality, it might improve overall code maintainability. In general JSTL might improve in comparison to scriptlets:

- JSTL are easier to test, maintain and read
- JSTL can be reused
- JSTL can just fail without breaking segments of the page

For these reasons JSTL would be a good way to replace the scriptlets.

While no feature is missing from the application (as defined by Description), a late suggestion by DTU brew house was to have a "flow diagram" maker. Although the functional requirements of this flow diagram puts it closer to a visualised spreadsheet detailing input/output of processes involved in brewing. As such this could be the focus of a future update of the application. However in order to avoid scope creep and due to this being suggested late in the implementation phase, it was decided to focus on admin functionality. Finally we may also consider creating a cookie for automatically signing in users, this in turn would however require some considerations about the security of the cookies.

8.2 Development method

The development method as mentioned in the results section appeared to be much better suited than envisioned to small amounts of parallel work. Compared to Scrum it provides less excessive planning, but does still seem to allow reasonable options of concurrent work flow. There remains still however the added risk of creating a critical path of work through the project planning. With this critical path in mind, it might be worthwhile to before hand identify it and minimise it if possible.

The other aspects such as iterations provide stability in ensuring the client will have a working software increment, regardless of substantial delays happening. Although this was not the case in this project. Additionally the structure of the development provided a strong red line between described features, use cases, functionality and the end-product.

Had the project been closer to real life in term of team size, Scrum might rightfully be a better choice, although sharing many ideals of Scrum, the latter still provides a better breakdown of work for larger teams. The development method provided no impediment to the accomplishment of the success criteria of the project, we may therefore assume it added to the success of the project rather than taking from it.

8.3 Alternate solutions

The biggest choice between solutions happened quite early in development, namely a choice between using pure JSP or moving some functionality to client side in form of javascript.

Monitoring of calls has shown that the uploaded data (for smaller recipes) is around 675-1000 bytes (although very much dependant on input sizes, number of ingredients, and size of recipe comments etc..). As a pure JSP solution would require more frequent round trip communication, this in turn might cause users with poor internet connections to experience latency issues (or lag), and as such javascript provided a more responsive and reliable solution.

Quite often in the project I considered applying some form of concurrency, however in all cases the operations where neither complex enough, or heavy enough to effectively apply multiple threads or processes. Clearly, one could have used multiple processes to calculate some characteristics of brewing, namely gravity points. However this operation is quite simple and not very computationally heavy either and as such would also make a poor appliance for current processes.

Some concurrency considerations can and should however be applied when dealing with some operations such as saving. Imagine a case where two users, at the same time creates a recipe with the same name. At some point in the code we must ensure only one of them gets to save their recipe. Specifically this is handled two ways, firstly in the database, one SQL query must be run before another and as such the one submitted first sits on the primary key. At this point when the other SQL command is run, it will return an SQL exception, as the primary key is already in use. An added check may be applied to ensure that the recipe name was successfully saved by a user, if so, the ingredients may be saved also.

While not extremely ineffective, MYSQL's "like" keyword may in the worst of cases need to search through all of a database. Performance might be improved somewhat by creating indexing on search keys. On the other hand MYSQL searches are well optimised and creating a search algorithm might be somewhat useless, unless the database is maintained sorted. Indexing might help improve this slightly. However the current implementation of searching might not benefit greatly from indexing as two wildcards "%SearchKey%" means a sequential search has to be performed (O(n) time). Rather a better focus here might be to limit searching to improve performance in a way as to make use of indexing as well.

8.4 Project management

Quite a few aspects of project management includes maintain control over a project, that is to say, in a project a certain number of things need to be controlled and other not so much. In the case of larger teams, the overall method might be a daily meeting to review progress, and keeping an eye on the time schedule and budget. During this project most basic aspects of control does not apply directly. Nonetheless maintain control of the schedule very much does, in this case the workload of operations was overestimated in the start of the project, however was adjusted progressively. Towards the end of the project the the time estimations for tasks was much more dead-on and thus control of the schedule was maintained.

During the project several methods of risk assessment was reviewed and considered, however in many cases they became very institutional in addition to identifying risks. As such it becomes clearly better to use a more general sort of risk assessment, that does not concern itself with institutional issues. Finally it might be worthwhile to even consider why project management is useful in projects. The four perspectives of project management, are of course interconnected[5]. The interconnection adding to the success of a project. The vision contributes to the complexity of the project and specifies

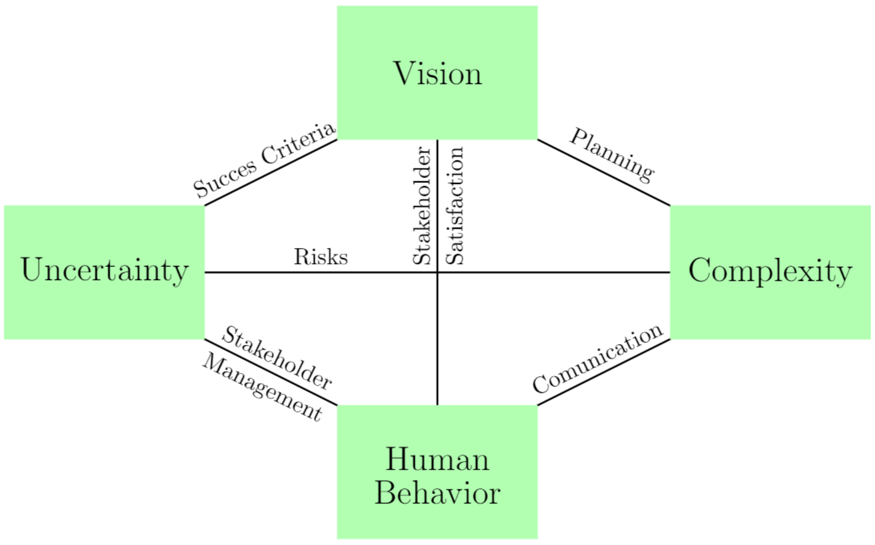


Figure 16: Relationship between aspects of the project management

success criteria, that in turn adds to the complexity as well. The complexity being determined by the scope, associated risks, the planning and the development methodology. Uncertainty in turn effects complexity and requires stakeholder management to ensure satisfaction of the resulting product. In any case it also requires flexibility, as to respond to changes presented by the stakeholders. Human behaviour affects the success of the project, failure to take the people involved into account increases the complexity, and adds to the uncertainty of the project. Though it can be noted, few aspects of human behaviour can be applied here, due to especially, the project team consisting of one person. The relationship between the aspects of management can be seen visualised in figure 16. Therefore using aspects of project management, ensures a very strong connection between the different aspects of a project.

9 Conclusion

An important point throughout the project has been to create an application which has the potential of being expandable, which both the database and application allows for. Another key point was to have an easily readable and usable GUI, and I am overall happy with the look and usability of the UI. In the end more time could have been spend perfecting and refactoring the model however functionally there are no problems. Furthermore tests have proved the correctness of output of the application layer. And our use case tests show functional correctness of not only the application layer but also shows the GUI as working.

During most courses at DTU, it is often the case that projects only contain a fraction of functional learnings, and as such it has been interesting working with the full array of design choices instead of only a select a few. The systematic start-to-end development has such provided a good insight into real life projects and application development processes. Furthermore having to make considerations such as maintainability and scalability of a close to full scale system has given a good insight into how applications might go about solving those issues. During the project, questions such as usability vs theoretical practises has shown a sharp contrast between the two and proven that often trade-offs are required and may be used practically, this being the case especially in the database design.

Reviewing our success criteria, namely stakeholder satisfaction, and the accomplishment of the vision as supported by our use cases. We may with some optimism declare the project a success given those criteria. With the use cases derived from the project description, desired functionality implemented, and with the satisfaction of both stakeholders namely; DTU brewhouse and Christian D. Jensen (in terms of functionality). Remembering the definition of usefulness discussed earlier and assuming the use cases describing a useful tool, we can declare end-product fully functional and accomplishing our success criteria.

Dan Roland Persson (s134837)

10 Glossary

- Fermentables: Contain the sugar the yeast uses to produce alcohol and carbon dioxide, these also add flavoring and color to brews.
- Hops: The flowers of the hop plant is a member of the hemp family. Hops contains oils with a very bitter flavor, this flavor is used to counter the sweetness from the malt to create balanced beers. Furthermore they also add to the aroma, freshness and preservation of beers.
- Flavoring Hops: Addition to hops, adds flavouring and aroma, usually added in the later states of the boil.
- Yeast: Yeast turns the sugar extracted into alcohol and carbon dioxide.
- Mash steps: Step mashing is for example a mash program in which the mash temperature is progressively increased through a series of rests.
- Spice: Various ingredients, an example could be vanilla, or Annis.
- Special steps: Any additional steps required for the brewing.
- Malt Extract: Liquid or dried variants exists of various malt sugars. Also known as the extracted fermentable sugars from malt barley.
- Mash - The hot water steeping process that promotes enzymatic breakdown of the grist into soluble, fermentable sugars.
- Steep: Set the grains in hot waters to release their flavors.
- Late: Fermentables added late in the process such as sugars.
- Wort: The combined extract and water mixture, sometimes referred to as unfermented brew.
- Priming solution: Adds priming sugars to the brew, which adds a small controlled fermentation to the bottled beers. The co2 given by this fermentation carbonates the brew.
- Fining agent: Removes some of the protein in the brew, making it clearer.
- Gravity: Is a measure of density.
- Original Gravity: The gravity of the wort before fermentation.
- Final Gravity: The gravity of the wort after fermentation has been completed.
- Ppg: Points per pound per gallon.
- ER diagram: Entity–relationship diagram.

- EER diagram: Enhanced entity–relationship diagram.
- Attributes: Also known as columns or fields.
- Tuples: Also known as rows or records.
- HTTP: The Hypertext Transfer Protocol (HTTP) is an application protocol for the Internet.
- JSP: Java server pages, can be used to create dynamic content for web pages.
- Servlets: A Java servlet is a Java program that extends the capabilities of a server.

References

[1] Brewgr.com. Homebrew Calculations. http://brewgr.com/calculations.

[2] Ray Daniels. Designing Great beers. Brewers publications, 2000.

[3] Oracle. Java Technologies for Web Applications. http://www.oracle.com/technetwork/articles/java/webapps-1-138794.html, 2006.

[4] OWASP. OWASP Top Ten Project. owasp, 2013.

[5] Harvey Maylor. Project Management, Fourth Edition. Pearson Education Limited, Harlow, England, 2010.

[6] Henry F. Korth Abraham Silberschatz and S. Sudarshan. Database System Concepts, Sixth Edition. McGraw-Hill Higher Education, 2011.

[7] Flemming Schmidt. Normalization Slides. 02170 Database Systems DTU compute, 2015.

[8] Hans Henrik Løvengreen. Basic Concurrency Theory. Mathematics and Computer Science Technical University of Denmark, 2015.

A Use-cases

A.1 Use case 1:

Name: Creating a brew
Description: A user wishes to create a simple brew with one of each type of ingredients and wishes to print the recipe.
Actor: A system user
Preconditions: None
Main scenario:

1. User navigates to the recipe designer.
2. User enters desired ingredients from lists of ingredients.
3. User exports to pdf and receives a download

Alternative Scenario:
3A- If an error exists in the created recipe a warning should pop up upon exporting.

A.2 Use case 2:

Name: Saving a recipe
Description: A new user wishes to create a simple recipe and save it for later use.
Actor: A user
Preconditions: Use case 1
Main scenario:

1. User saves the recipe
2. User is prompted to create an account
3. User creates the account
4. User saves the brew
5. User has a menu option to review saved brews.

Alternative Scenario:
2A- User can avoid making an account but can't save brews.

A.3 Use case 3:

Name: Publishing brew.
Description: A user wishes to publish his/her brew such that other users can find it.
Actor: A user
Preconditions: A created recipe (see use case 1/2)
Main scenario:

1. User gives his/her recipe a name
2. User publishes the recipe from the designer
3. User can locate his/her recipe from the search function

Alternative Scenario:
1A- Name of brew already exists user should be prompted for a new name.
1B- User enters no name, system should prompt user for a new name.
1C- Brew contains missing ingredients or errors, user will need to fill these in first.

A.4 Use case 4:

Name: Missing ingredient
Description: A has a custom ingredient such as a special hops, which is not present in the database
Actor: A user
Preconditions:
Main scenario:

1. User clicks the option add custom ingredient
2. User is prompted for a name
3. User can set related attributes for the ingredient
4. System handles the ingredient correctly.

Alternative Scenario:
None

A.5 Use case 5:

Name: Register
Description: A new user wishes to register himself to the website.
Actor: A user
Preconditions: None
Main scenario:

1. User presses the register button
2. User fills in required information
3. User can now login

Alternative Scenario:
1A- prompted information is incorrect, error message displayed.
1B- Email or user name already exists in the system, error message displayed.

A.6 Use case 6:

Name: Load saved recipe
Description: An existing user wishes to load a previously saved recipe and edit it, saving the changes.
Actor: An existing user
Preconditions: An existing recipe
Main scenario:

1. User checks his/her saved recipes
2. User clicks desired recipe
3. Recipe is now editable and an ingredient is added
4. User saves the changes
5. Changes now in effect

Alternative Scenario: 1A- User is not logged in and will be prompted to do so.

A.7 Use case 7:

Name: Recipe search
Description: A user wishes to search for a brew recipe
Actor: A user
Preconditions: Some existing recipes
Main scenario:

1. User navigates to the search area
2. User enters either a name or a brew type
3. System show a list of possible recipes and some information about them.
4. User can pick one and is shown the full recipe

Alternative Scenario:
3A- System has no brews matching the description.

A.8 Use case 8:

Name: Scale recipe
Description: A user wishes to know how many ingredients he/she requires to up a brew batch from ex. 1L to 5L
Actor: A user
Preconditions: An existing recipe
Main scenario:

1. User scales the recipe size to the desired value
2. Numbers related to the recipe are updated.

Alternative Scenario:
None

A.9 Use case 9:

Name: Adding ingredients
Description: A user is creating a recipe with a target variable (for example color)
Actor: A user
Preconditions: An existing recipe
Main scenario:

1. User adds a Fermentable
2. User adds a hops
3. User adds a yeast
4. User does not hit his/her target brew color and begins to change variables to achieve this
5. The estimates for the beer should update live as the user makes changes

Alternative Scenario:
1A- User inputs a negative variable, system should handle by not allowing negative values to effect the calculations

A.10 Use case 10:

Name: Adding a comment on a recipe
Description: User wishes to add a comment on a recipe
Actor: A user
Preconditions: An existing recipe
Main scenario:

1. User navigates to the recipe
2. User supplies the comment input
3. User clicks submit
4. The comment can now be found by other users.

Alternative Scenario:

A.11 Use case 10.1:

Name: Delete comment
Description: User wishes to delete his comment
Actor: A user
Preconditions: An existing recipe and comment
Main scenario:

1. User navigates to his inventory
2. User can now locate his/her comment
3. User clicks delete comment
4. The comment can no longer be found by other users.

Alternative Scenario:

A.12 Use case 11:

Name: Remove recipe
Description: An admin wants to remove a recipe
Actor: An Admin
Preconditions: Some existing recipes
Main scenario: An Admin wishes to remove a recipe

1. Admin locates the recipe via the search function
2. In the search menu the admin has the option to delete the recipe
3. Admin clicks the delete button and the recipe is removed by the system.

Alternative Scenario:

A.13 Use case 12:

Name: Remove Comment
Description: An admin wants to remove a comment on a recipe
Actor: An Admin
Preconditions: An existing comment on a recipe
Main scenario: An Admin wishes to remove a recipe

1. Admin navigates to the recipe containing the comment.
2. Admin has the option to delete any comment.
3. Admin clicks on the comment he wishes to delete and the system deletes it

Alternative Scenario: If the signed in user is not an admin, he does not have the option to delete comments

A.14 Use case 13:

Name: Add article and delete
Description: An admin wishes to create a new article
Actor: An admin
Preconditions: An existing comment on a recipe
Main scenario:

1. Admin navigates to the admin menu
2. Admin clicks on the article section
3. Admin Inputs the needed information and clicks add
4. Article can now be found on the front page.
5. Admin now deletes the 2nd newest article (which no longer shows up on front page)

Alternative Scenario:

A.15 Use case 14:

Name: Delete user
Description: An admin wants to remove a user from the system
Actor: An Admin
Preconditions: An existing user
Main scenario:

1. The admin navigates to the admin menu and reviews users.
2. The admin finds the user and clicks delete
3. The user has been removed from the system.

Alternative Scenario: User is an admin and cannot be deleted.

A.16 Use case 15:

Name: Manage default ingredients/beer types
Description: An admin wants to delete some default ingredients/beer types and add some new ingredients/beertypes
Actor: An Admin
Preconditions: Existing beertypes and default ingredients
Main scenario:

1. The admin navigates to the admin menu and reviews ingredients.
2. The admin can delete and add ingredients and beertypes as he wishes.
3. Depending on the admins queries the system should handle the changes.

Alternative Scenario:

A.17 Use case 16:

Name: Password Recovery and change account info
Description: A user has forgotten his/her password and wishes to recover his/her account
Actor: An User
Preconditions: An existing user
Main scenario:

1. In the login menu user clicks and navigates to the password recovery.
2. User inputs his/her email linked to the account.
3. User checks email account and fines new password in the mail
4. User logs in with the new password and navigates to his/her account settings
5. User now changes his/her password to the desired password
6. Password can be used to login from then on

Alternative Scenario: 5A - Password does not match requirements for passwords, an error should appear.

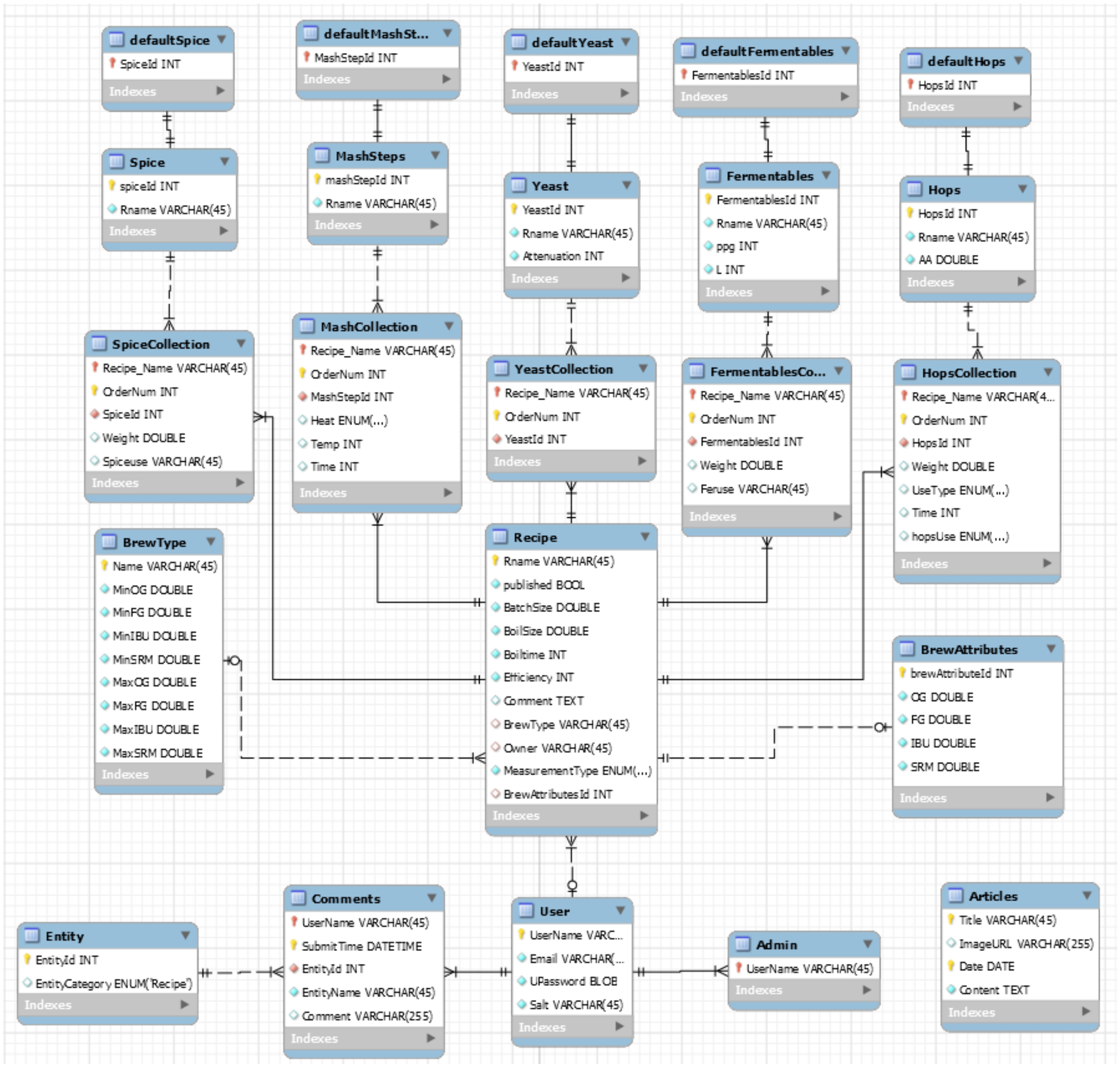
A.18 Use case 17:

Name: Switch email
Description: A user wishes the change the email linked to his/her account
Actor: A User
Preconditions: A existing user
Main scenario:

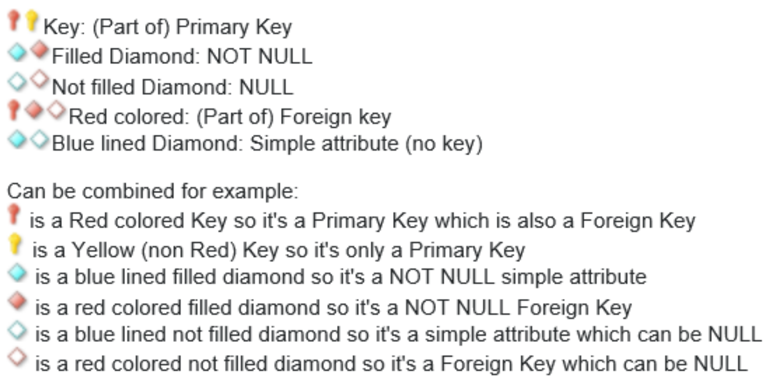
1. User navigates to his/her account section when logged in
2. User can now enter the new email
3. Recovery mails will be send to this mail from now on.

Alternative Scenario:

B Database Models



B.1 ERR icons explanation



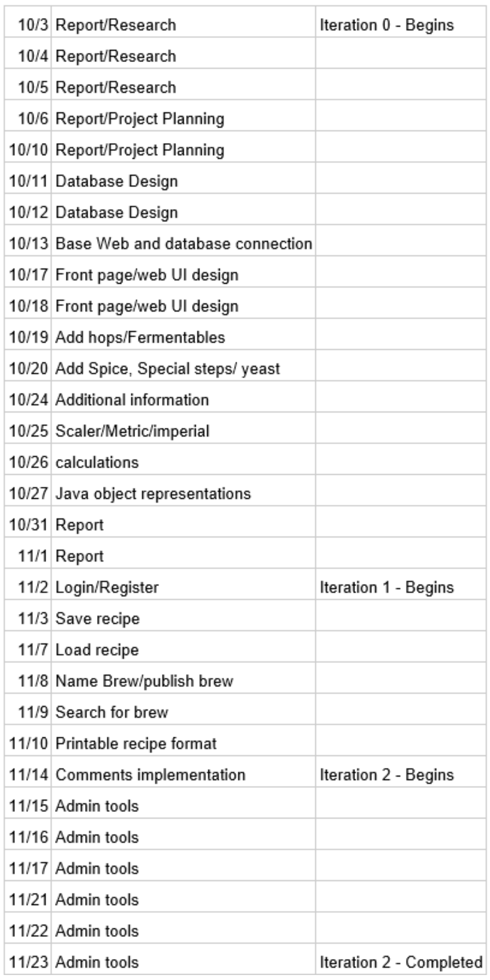
🔑🔑 Key: (Part of) Primary Key
◆◆ Filled Diamond: NOT NULL
◇◇ Not filled Diamond: NULL
🔑◆◇ Red colored: (Part of) Foreign key
◆◇ Blue lined Diamond: Simple attribute (no key)

Can be combined for example:
🔑 is a Red colored Key so it's a Primary Key which is also a Foreign Key
🔑 is a Yellow (non Red) Key so it's only a Primary Key
◆ is a blue lined filled diamond so it's a NOT NULL simple attribute
◆ is a red colored filled diamond so it's a NOT NULL Foreign Key
◇ is a blue lined not filled diamond so it's a simple attribute which can be NULL
◇ is a red colored not filled diamond so it's a Foreign Key which can be NULL

Explanation of the EER color and symbol scheme[12].

[12]http://stackoverflow.com/questions/10778561/what-do-the-mysql-workbench-column-icons-mean

C Project-plan



10/3	Report/Research	Iteration 0 - Begins
10/4	Report/Research	
10/5	Report/Research	
10/6	Report/Project Planning	
10/10	Report/Project Planning	
10/11	Database Design	
10/12	Database Design	
10/13	Base Web and database connection	
10/17	Front page/web UI design	
10/18	Front page/web UI design	
10/19	Add hops/Fermentables	
10/20	Add Spice, Special steps/ yeast	
10/24	Additional information	
10/25	Scaler/Metric/imperial	
10/26	calculations	
10/27	Java object representations	
10/31	Report	
11/1	Report	
11/2	Login/Register	Iteration 1 - Begins
11/3	Save recipe	
11/7	Load recipe	
11/8	Name Brew/publish brew	
11/9	Search for brew	
11/10	Printable recipe format	
11/14	Comments implementation	Iteration 2 - Begins
11/15	Admin tools	
11/16	Admin tools	
11/17	Admin tools	
11/21	Admin tools	
11/22	Admin tools	
11/23	Admin tools	Iteration 2 - Completed

The additional time unaccounted for in the project plan, has been spend testing, refactoring and writing report.

D Project description

Students: Dan Roland Persson s134837

Project ECTS points: 15

Supervisor: Christian Damsgaard Jensen

Project start date: d. 3 October 2016

Turn in date: d. 3 January 2017

D.1 Description:

The vision of the project: “To create a useful tool for recipe design for homebrewers and microbreweries” While the mathematics of brewing does not ensure a great tasting end-product, it can help scale recipes. It can predict colour, bitterness, alcohol and other aspects of a potential brew.

A user of the system should be able to enter his/her ingredients and some aspects of the process and be able to see the results of that recipe, the system should of course also allow for saving recipes, publishing and for printing them in an orderly fashion.

The system as such would consist of three major parts, the database, the application layer and a web-based GUI. The database should include such things as different ingredients, user information and recipes among many more things. The database is to be designed with thoughts of future extensions of functionality and be structured in such a way as to minimize redundancy.

The application layer of the system should be well tested by automated test cases to ensure the correctness of functionality. The web part of the project should include an easily navigable website, with the possibility of creating an account and commenting on others recipes.

The construction of the system should draw on software engineering and project management principles to ensure a logical construction of the project.

The database system will be coded in MySQL. The application layer in java The web frontend in HTML-JavaScript and in either JSP or Php.

Dan Roland Persson (S134837)

E Deployment

In this section we will quickly go over the steps required to deploy the system. Overall an MYSQL server must be setup on port 3306 and for the current version, a root user with the password "tukanlobo19". In case of the system being deployed on a Linux machine the SQL server must be setup to be case-insensitive for SQL commands, as is the case by default on Windows. Additionally it might be worthwhile to ensure the database is not accessible outside of the local machine. Once completed the creation and population files may be uploaded to the machine and run via SQL commands.

In addition to this a tomcat 8 server must be deployed and run on the system. At this installation step it might be worthwhile changing the tomcat admin page settings such that only one IP may access the tomcat administration root page. Once the server is running, the admin page will allow for deployment of the project war file. Additionally the following pages give a very specific and in-detail description and guide to the respective setups (in order):

- https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-16-04
- http://dba.stackexchange.com/questions/59407/how-to-make-mysql-table-name-case-insensitive-in-ubuntu
- https://www.digitalocean.com/community/tutorials/how-to-install-apache-tomcat-8-on-ubuntu-16-04

It should be noted that new administrative users may only be promoted through SQL and as such must be promoted/removed through the command prompt on the hosting server.