



Erhverusakademi København

Hovedopgave/Final exam

exam 16.Januar

Prædefineret information

Startdato:	15-11-2017 09:00	Termin:	jan 2018
Slutdato:	20-12-2017 12:00	Bedømmelsesform:	Dansk 7-trinsskala
Eksamensform:	Mundtlig prøve	ECTS:	15
SIS-kode:	0118 30501795A000 70050 - MDT AF 7TRIN		
Intern bedømmer:	Christian Kirschberg		

Deltager

Navn:	Dana-Maria Iliescu
Kandidatnr.:	9724 5891 jan 2018 1192 3966
UNI-C ID:	(Ikke sat)
KEA-id:	dana0396@kea.dk

Information fra deltager

Fødselsdato *: 02.10.1996

Opgaven må gøres Ja
tilgængelig i digital eller
analog form for
studerende og ansatte på
KEA via KEAs biblioteks
opgavearkiv:

Opgaven må gøres Ja
tilgængelig for alle andre
i digital form, via en
internetportal drevet af
KEA eller på anden
teknisk vis, som svarer til
dette:

Indeholder besvarelsen Nej
fortroligt materiale? *:

Gruppe

Gruppenavn:	Hamamas
Gruppenummer:	25
Øvrige medlemmer:	Petru Birzu, Maciej Tilma

KEA - Copenhagen School of Design and Technology
Computer Science, DAT 15I



Hamamas Final Project Report

Students

Dana-Maria Iliescu, 02.10.1996

Maciej Tilma, 09.11.1988

Petru Bîrzu, 15.08.1996

Lecturer

Christian Ole Kirschberg

Date

20.12.2017

Table of Contents

1. Introduction	1
1.1. Aim	1
1.2. Problem formulation	1
1.3. Vision	2
1.4. Mission	2
1.5. Values	2
2. Business analysis	3
2.1. Stakeholder Analysis	3
2.2. Project management	6
2.2.1. People	6
2.2.2. Project	6
2.2.3. Process	7
2.2.4. Product	7
2.3. Competitor analysis	7
2.3.1. Current Competitors	7
2.3.2. USP	9
2.3.3. ESP	9
2.3.4. SWOT	9
2.4. Target audience	10
2.5. Feasibility study	11
2.5.1. Operational Feasibility	11
2.5.2. Technical Feasibility	12
2.5.3. Schedule Feasibility	12
2.5.4. Legal Feasibility	13
2.5.5. Political Feasibility	13
2.5.6. Economic Feasibility	13
2.6. Risk Management	13
2.6.1. Risk Analysis	14
2.6.2. RMMM Plan	15
3. Technical analysis	16
3.1. Software engineering	16
3.1.1. Code standards	16
3.1.2. Engineering methodology	17
3.1.2.1. Agile vs Non Agile	17
3.1.2.2. XP & Scrum	20
3.1.3. Team structure and roles	21

3.1.4. Use Cases	22
3.1.5. Prototypes	25
3.1.5.1 Paper Prototype	25
3.1.5.2. Wireframes	26
3.1.5.3. Mock ups	28
3.1.6. User stories	30
3.1.7. Style guide	30
3.1.8. Minimum viable product	32
3.1.9 UML	33
3.1.9.1. Architecture (overview)	33
3.1.9.2. UCD	34
3.1.9.3. SSD	34
3.1.9.4. SD	35
3.1.9.5. Database model	36
3.1.10. React Components	37
3.1.11. Sprint documentation	38
3.1.12. Quality concepts	43
3.2. Software development	46
3.2.1. Technologies	46
3.2.2. Progressive Web App (PWA) vs. Native App vs. React Native	48
3.2.3. Software Configuration Management	49
3.2.4. User testing	53
3.2.5. Code snippets	54
3.2.6. Maintenance	58
4. User Manual	60
5. Usability Heuristics Evaluation	64
6. Possible improvements and new features	67
7. Reflections	69
7.1. How Scrum + XP worked	69
7.2. Professional & Personal development	69
8. Conclusions	70
8.1. Product Conclusion	70
8.2. Process Conclusion	70
Special thanks	71
Bibliography	72
Appendix	73

1. Introduction

This report encapsulates the work and learnings from the Hamamas team during the development of a web application. In this document you will be able to find all the information regarding our project management, design and development. We have chosen to create the report in the following manner. First off you will read about the project itself. Why did we create Hamamas, what is the goal with our software application, are there others that do the same and why is Hamamas so unique. Following our more business related chapters we focus on software design and engineering practices. In these chapters we touch upon how we managed our processes, what did we do to ensure quality, flexibility and solid teamwork. After the design chapters we address the development. In these chapters we talk about the technologies that we have used, why we used them and which aspects of the code are interesting enough to highlight. To conclude the report we decided to write down our learning moments, future improvements and conclusions of what went well and what we could improve upon. We hope you enjoy reading about our work, as much as we enjoyed creating our Hamamas application.

The story behind the name

While searching for a suitable name a lot of options came up, ranging from acronyms and short sentences to word combinations. Eventually we decided to look up “Enjoy your meal” a.k.a. “Bon Appetit” in as many languages as possible. We found a website called omniglot (*Omniglot is an online encyclopedia focused on languages and writing systems. The name Omniglot comes from the Latin prefix 'omni' and the Greek root 'glot'¹ which means Omni; possessing all of its class, glot; 'to have a tongue' So, possessing all tongues, but in this case it is languages.*) While browsing through its many languages we found the word Hamamas, which means “ Enjoy your meal” in Tok Pisin, an English-based Croele language spoken in Papua New Guinea. Not only did we like the sound of the word, it also encapsulated everything we wanted to communicate in just one single word so from there on we called ourselves Team Hamamas.

1.1. Aim

The aim of our project was creating a piece of software that will give value to a user and show our skill and proficiency in software engineering, software development and the business aspects that come along with creating a project.

1.2. Problem formulation

As a team we all agreed that there was a need for a fresh and new approach to showing recipes, therefore we formulated the problem we would like to solve as the following:

With our project we try our best to address the lack of easy and fun recipe generators.

¹ <https://www.omniglot.com/language/phrases/bonappetit.htm>

1.3. Vision

When thinking further towards the future we see our application being able to follow this vision:

Using technology to increase people's ability to cook and decreasing food waste in the world by making cooking easy and fun.

1.4. Mission

Our idea from the start eventually became the mission of Hamamas, which is the following:

Making cooking fun, easy, fast and decreasing food waste in the process.

1.5. Values

Besides being purely a software product, we believe that more people should learn to cook and have fun doing it. In current day society cooking is becoming something that is regarded as time consuming and boring. We hope that by making this application we can prove the opposite. Besides not cooking often, people tend to buy a lot and not finish it before its expiration date, this just wastes their time going to the grocery store, their money and of course, food.

Our values are the following, in no particular order:

Easy to use:

We believe that Hamamas goes the extra mile in order to give its users an experience that has not been made easy so far. Just showing pictures and limiting to swipes for choice, it cannot get easier than that.

Entertaining:

We present food in its most entertaining and convincing way possible, by using pictures. This should encourage a fast choice of the user and also give them a visual of their end goal, which in this case will be cooking a meal.

Educational:

The application will show exactly how to cook the chosen meal, this is its most powerful aspect. Every user will be able to learn by doing and reuse that knowledge in further recipes. This means the more the user uses the application, the better they will get in cooking.

No wastefulness:

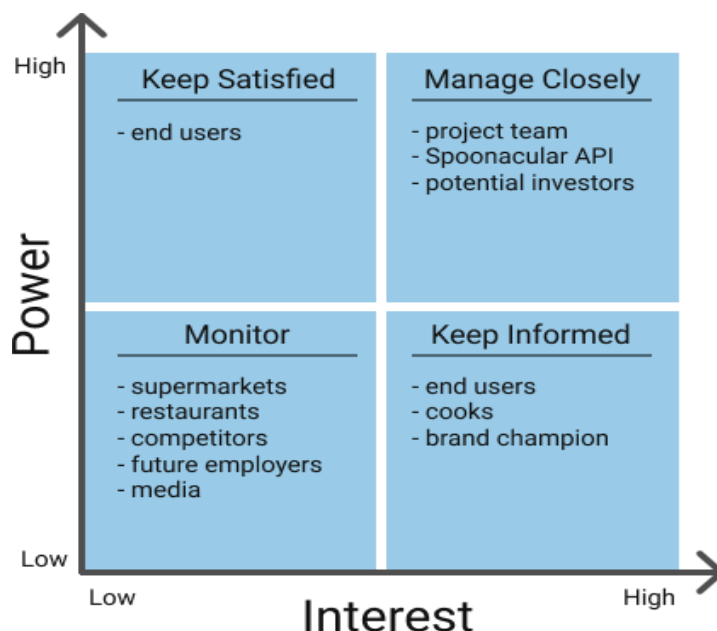
It is of course just an app, we will not decrease food waste directly. But it is a fact that when people tend to cook more they waste less. On top of that they will use their hard earned money on something else instead of buying food and just letting it go to waste.

2. Business analysis

In the following chapters we will present our business research, which encompasses different topics such as stakeholders, the four Ps (People, Project, Process, Product), risk management, SWOT, feasibility analysis, and our competitors.

2.1. Stakeholder Analysis

As each one of the stakeholders is either affecting or affected by our project we needed to know how to deal with them. First we brainstormed about who can be involved in this and then we prioritized them, as seen in the figure below.



Power/Interest grid for stakeholders

After doing so, we analyzed every stakeholder to see exactly how they can influence our project, what is there at stake for them, if there is any negative impact they could have, and what we could do to prevent that to happen. This can be viewed in the table below.

Stakeholder	Power	Interest	Concerns	How to handle
Project Team	High Power Steers the direction of the project, makes requirements, develops, and has the power to stop the project	High Interest Wants to deliver a good useful piece of software	Unavailability of one or more team members, loss of motivation, misunderstandings which waste time and effort	Good communication and regular meetups, keep each other up-to-date with their work progress and availability so that the necessary actions can be taken

End Users	High Power They are the ones using the application, without them the product would be useless	High Interest Want to have an easy to use and fun recipe generator	Not satisfied with the UI, UX, speed and/or generated recipes	Have some users for testing, keep them informed about feature changes in the project so they can give feedback
Spoonacular API	High Power Provides us with the recipe data and the search by ingredients algorithm	High Interest They gain popularity and exposure, making other people want to use the API	Can change their subscriptions, influencing how many requests/month we are allowed to make	Have an agreement with them about a fixed subscription before we start
Potential Investors	High Power With their investment we can upgrade our API subscription so we can provide more requests	High Interest If our product is successful they will benefit as well	Can choose to invest in a project of a bigger scale, if ours doesn't shine	Ensure effective communication and inform them about our project, thus they see the progress and dedication
Cooks	Low Power Using the application	High Interest The application could potentially help them improve their skills, find inspiration for new recipes, experience with different ingredient combinations	Might choose the traditional approach of cooking books over a software product	Keep them informed about our features and the ease of use, conduct user tests
Brand Champion	Low Power Doesn't have power on the project, cannot stop it or influence its progress	High Interest Is passionate about and believes in our application	Can lose motivation, thus cannot be "the living embodiment of the brand promise" ² anymore	Inform at all times about the project status

² <https://www.forbes.com/sites/work-in-progress/2011/07/12/be-your-own-brand-champion-or-get-one-now/#3e0f5d564b43>

Supermarkets	Low Power Doesn't have authority over the project	Low Interest Even if the idea is that the user inputs ingredients which he already has, sometimes there are more ingredients needed. In this case the supermarkets would benefit as users will buy the missing ingredients	There are no concerns regarding them	We do not need to handle them in any way
Restaurants	Low Power Can influence the product in the future, if they decide to integrate parts of it, for example by using their API or database for recipes, so clients can decide what to order based on ingredients	Low Interest Restaurants usually use menus for displaying their available dishes, so most of them will show little interest in the project	Might think that it is not necessary for them to use the app instead of the menus	Make sure they are aware of the project and the possibility to turn a recipe generator into a way for their customers to see quick and easy which dishes they offer based on ingredients
Competitors	Low Power Don't have authority over the project	Low Interest They are only interested to know what kind of projects come on the market	They can be so much better that people will choose them over our application	Might get inspiration from them
Future employers	Low Power Don't have authority over the project	Low Interest They are not directly interested in the project promise, but more in how we built the product and which technologies we used	They cannot see the quality of the code as it is closed source	Make sure the application runs smoothly and bug-free so they can see the value of the code

Media	Low Power Doesn't have authority over the project	Low Interest To reveal a new recipe generator application (on websites that are food related), thus they get more readers	Might think project is not big enough or doesn't have innovative character	Promote the application through our network and social media so it gains visibility online
--------------	---	---	--	--

Stakeholder analysis table, describing power, interest, concerns and how to handle for each stakeholder.

2.2. Project management

Effective project management is broken down into four P's: people, project, process and product. We will address all of these in as it follows.

2.2.1. People

Identifying the roles people play in our project is the first step to a success. People are our primary resource for this project, and a well-conducted team can greatly increase the chances for success.

We have our team for starters, all with their specific strengths. Besides the team we have some stakeholders that make this project into a success or failure; among the stakeholders we have: the services that we use and our users as the main stakeholders (for more in depth information we refer you to the Stakeholder Analysis chapter on page 3). We made the choice to give every team member equal responsibility, meaning full transparency and no hierarchy of any kind. Anyone from the team was able to come up with new ideas, approaches and review any task. This approach empowered our strongest asset and gave us the ability to bond as a team.

2.2.2. Project

The Hamasas project started in the first week of November and its MVP development ended in the first week of December. Understanding where our project can go wrong and how to avoid those problems will lead to a successful project. We track our progress on the Trello Scrum board and GitHub, we have daily updates and review each other's work to ensure the project moved forward and had the right momentum.

By reflecting on previous projects we decided to adapt Reels³ common sense approaches to software projects.

- *Start on the right foot.* Since we were an agile team we had the needed authority and autonomy in order to build this product.
- *Maintain momentum.* Self chosen subject and challenging work, personal and professional development motivated the team throughout the project.

³ Reel, J., "Critical Success Factors in Software Projects", *IEEE Software*, May 1999, pp. 18-23

- *Track Progress.* Progress was tracked using Trello and GitHub.
- *Make smart decisions.* Component based programming with React made a lot of the code base reusable or easily adaptable.
- *Conduct a post mortem analysis.* We did this by using the Sprint Retrospective meetings and the pull requests code integration strategy.

2.2.3. Process

After finishing multiple projects we know that a selection of the proper process for a project is a task that cannot be overlooked.

Choosing for an agile and test driven development was not just coincidence (for more information please refer to the methodology chapter on page 17); after assessing pros and cons and creating a workflow that fitted our team size, project complexity and deadline, we could be confident that we were able to finish the project in a desired state.

2.2.4. Product

At the beginning of our process the product wasn't perfectly defined. It was our task to establish and define the minimum scope of the product also known as the minimum viable product (MVP).

We wanted to empower users to cook. By giving them access to recipes in the easiest way possible, our product should become everyone's number one source for cooking inspiration and education in cooking. The core functionality lies in the applications ability to access a large set of recipes and deliver results based on ingredient search in a fast and reliable manner.

The scope for the first release was the following, a user should be able to sign up/login, then search for desired recipes based on ingredients of choice. Besides that, the user should be able to add to favourites any recipe and should be able to see the history of the search for backtracking. Another feature would be to change their password. Of course all of these features should pass acceptance criteria, inhouse testing, reviews and user testing.

2.3. Competitor analysis

In order to find out if our application is worth making, one of the factors we need to consider is what is already being offered and how we distinguish ourselves from the competitors. We have conducted an analysis of known/biggest competitors in the food application market. Having this overview gave us the ability to choose the right features to develop and to learn from what is already out there.

2.3.1. Current Competitors

The idea for Hamamas came from the lack of easy and approachable recipe generating applications for a mobile device (be it Android or iOS). Of course there are already similar

applications that might compete with our idea, hereby a shortlist of top searches from Google, Apple App Store and Google Play Store (Android).

Top Google Search

Supercook

- pros: High amount of recipes
- cons: Not focused for mobile and not created for ease

Foodwise

- pros: Focused on reducing food waste
- cons: Not made for mobile and adding ingredients is tedious

MyFridgeFood

- pros: Ability to add own recipes
- cons: Not made for mobile, UI is outdated and not user focussed

Top Apple - AppStore

Yummly

- pros: Beautiful UI, perfect display of recipes, designed for mobile on all platforms (iOS, Android)
- cons: No social aspect, recipes not marked based on difficulty

Kitchen Stories

- pros: Great amount of videos showing the cooking process, good UI design
- cons: Only for iOS, not made for speed

Tasty

- pros: A lot of filter options, build-in cooking timer
- cons: Focused on people who know how to cook, iOS only.

Top Google Play Store (Android)

Yummly

- pros: Beautiful UI, perfect display of recipes, designed for mobile on all platforms (iOS, Android)
- cons: No social aspect, recipes not marked based on difficulty

Cookpad

- pros: Social aspect with recipe sharing
- cons: Tablet focused, not usable on smaller devices

Foodies

- pros: Easy to use
- cons: Results don't display exact matches (bug in search function), no check for allergies

We have chosen only to specify the most obvious searches since our potential users will search for applications in a similar fashion. Based on that we can see we determined that

Yummly is our biggest competitor. With Yummly being both for iOS and Android, having a great UI and established user base, it will be our future social features and speed of usage that differentiate Hamamas from its main competitor.

2.3.2. USP

When we first came up with the idea, we also had a few Unique Selling Points (USP's) in mind that really stood out from similar applications. We have noted down the following:

- Mobile focused for all types and fast (not just linking to other sources but using data)
- The ability to choose recipes based on a familiar and simple concept of browsing through pictures.
- The possibility to connect to others that cook the same type or exactly the same food.

With these USP's in mind we were able to create most the of the Use Cases that can be read further down in the report in the Use Cases chapter on page 22.

2.3.3. ESP

When defining unique selling points we also discussed what our application could provide from an Emotional Selling Points perspective.

- It teaches its users to cook based on their favorite ingredients
- It connects users with each other through a common goal
- It becomes a personal cookbook for every user

2.3.4. SWOT

We are definitely not the first in creating a ingredient based recipe generating application. In order to visualize our current position we decided to put all the information available to us and summarize it into a SWOT matrix to have a quick and useful overview of our current state within the market.

This matrix reflects our product against others in the current market. Based on internal and external influence we can determine the likelihood of a successful launch to the market.

	Strengths <ul style="list-style-type: none"> • Easy to use • Designed for mobile, usable on all platforms • Only shows a near perfect / perfect recipe match 	Weaknesses <ul style="list-style-type: none"> • Needs API to function correctly • Relies on external data • Not the biggest amount of recipes out there
Opportunities <ul style="list-style-type: none"> • Less time to cook • Less wasteful society • Social food sharing 	Strength-Opportunity strategy <ul style="list-style-type: none"> • Ease of use will add value to speed of cooking • reduces 	Weaknesses-Opportunity strategy <ul style="list-style-type: none"> • with less time to cook there is no need for long sessions

	wastefulness	<ul style="list-style-type: none"> no need for huge amount of data, app serves well with current amount.
Threats <ul style="list-style-type: none"> others adding social features others using a simple picture based search market could be saturated inconsistent data from API 	Strength-Threats strategy <ul style="list-style-type: none"> Being ahead with ease of use makes copying by others not worth the time Features based on user base, that is hard to recreate 	Weaknesses-Threats strategy <ul style="list-style-type: none"> Getting own API for recipes Increasing stickiness of user by adding compelling features

SWOT analysis matrix

2.4. Target audience

When we started with the project, we defined a target audience that we want to approach with our application. Without certain users in mind it is near impossible to create an application that will be widely used.

Target group definition

Our application will serve anyone with an interest for cooking with the ingredients they know and have. Besides the cooking aspect it will serve as a social hub for people with similar cooking habits. This means that the demographic is really wide and therefore the target group is fairly large. With such a wide variety of potential users it's possible to have a substantial amount of users within a small period of time, because the pool is large enough to be shared with competitors. To illustrate an average user we decided to demonstrate three common users that we think fit our target group; you can read those in the user profiles below.

User profile #1: Dana, in her twenties

Dana is a student and part time employee in a big and social firm, really into meeting up with her friends and surprising them with a home cooked meal. She has been using a lot of apps to help her out with finding new recipes. She has been cooking for a long time so she feels confident to cook anything that looks delicious to her.

She used her tablet a lot in the kitchen because most free cooking applications were made for a desktop and aren't mobile friendly so she needed that bigger screen. Besides that it took her a while to find a recipe because the list generated from her searches were way too long and most of the meals were not up to her standards because the recipes required a lot more ingredients than she had at home.

She would like to have a solution that helps her save space in the kitchen, so no more tablet, and that generates recipes of ingredients that she actually has. On top of that it should be fast and not overwhelming.

User profile #2: Petru, mid-thirties

Petru is a high functional professional, always on the run and has no time to waste. He likes to cook because it breaks up his day and makes him calm, but of course it can't take too long and he doesn't want to head out to do groceries every time he makes something for dinner.

He has tried a lot of solutions, even paid for mobile apps. But all of them required too much time to find a recipe or were not made for all off his devices, he would like to have a small assortment of recipes shown to him based on what ingredients he has and just quickly choose based on cooking time and difficulty so he can get to cooking fast and enjoy the process more than searching for it.

User profile #3: Mac, mid twenties

Mac is a student, he is not too fond of cooking especially since he doesn't like to spend too much money on food. Everytime he cooks, he does it to empty out his fridge and his meals aren't the biggest success a lot of the times.

He has been looking into apps that could help him create something delicious out of his leftovers, but it just takes him too much time and it's sometimes frustrating that some recipes require even more ingredients than he put into his search. His perfect solution would be an application that would show recipes that match completely with his search and just display the result in a easy way. On top of that he wouldn't mind eating together with others that share his tastes, saves him from cooking so he can focus on other things.

2.5. Feasibility study

Before turning our idea into an actual project and put all our time and effort into it we had to answer the question: is it actually feasible to do this? GO or NO GO? In order to assess if the project was going to be successful or not, we analyzed the operational, technical, schedule, legal, political, economic factors.

2.5.1. Operational Feasibility

Since this project is carried out by us, a group of three students, and since we all like the idea and are determined to do this, we can affirm that it will be pretty easy to integrate this product into the group's "human systems"⁴. There won't be any difficulties for other people within the same company (if it were to be in a company) embracing the product or for anyone being against it, and it will not affect the organizational structure.

On the other hand, the product will help us achieve our vision: to have a simple and fun to use recipe generator based on ingredients that you have at home and for each user to be able to create their own little cookbook in no time.

⁴ <https://www.madebymany.com/stories/operational-feasibility>

2.5.2. Technical Feasibility

We have the necessary hardware (computers for developing, phones for testing wireframes), software (software for coding, version control, code storage, mock-ups and wireframing, scrum board, communication)⁵ and other required technologies (Internet, an API for getting the recipes).

There were however some risks factors that we had to take a look at:

1. project size: **medium**
2. project complexity: **high**
3. developers' familiarity with the type of application and technology: **medium to high**
4. users' familiarity with the system type: **medium to high**

Let's go into detail on this for a bit. Considering all the requirements and features that we decided on we can say that the project size is medium and complexity is high, noting also that the project duration was 4 weeks. We are familiar with the type of application which is a web application, since we have carried out similar projects in the past, and we worked with Node and MongoDB before, React being the only technology we didn't have so much experience with. Our end users will be all kinds of people: us, students, cooks, moms, basically anyone with a connection to the internet can use it. We consider that these people are familiar with the system type, as they have been through this process before: you sign up to an application, log in, input some information and choose what you desire from the output (choose which recipe you'd like to see). This would be medium familiarity. High familiarity would be for Tinder users. Swipe left if you don't like the recipe picture/title, swipe right if you like it and want to see the cooking instructions.

In conclusion, the overall risk is very low, and the project is technically feasible due to the team's knowledge and access to hardware and software.

2.5.3. Schedule Feasibility

We could say that for us carrying out the schedule feasibility before starting with the project made the difference between failure and success. We started out with a ton of cool ideas for features we would have loved to implement (filters for vegan, spicy or gluten-free recipes to mention just a few), but the schedule was very tight and we were a group of 3 people. We planned 4 one-week-sprints which were easily filled out with the core functionality, as based on experience and previous projects we knew what we were capable of and in how much time. So we got to the conclusion that it is only feasible to do this if we stick to the core functionality for the first release of the product, otherwise we will end up with an unfinished product and frustration that we could not reach the point we wanted. We kept in mind that we can always add to it and improve it in the future.

Sprints	1st Sprint (05.11. - 11.11.)	2nd Sprint (12.11. - 18.11.)	3rd Sprint (19.11. - 25.11.)	4th Sprint (26.11. - 02.12.)
What to do	- setup work environments - requirements	-software design - UI design - register feature	-search view - results view - cooking instructions	-profile view - navigation - routing

⁵ You can see specifically all the technologies we have used in Technologies section, Software Development chapter.

	- use cases - user stories	- login feature	view - top bar	- favorites view - history view
--	-------------------------------	-----------------	-------------------	------------------------------------

Basic schedule for the project. A detailed sprint documentation can be viewed in the Sprint Documentation section

2.5.4. Legal Feasibility

All technologies we are using are open-source, free to use for everyone, except the Spoonacular API which we pay a subscription for. Considering this, there is not legal issue stopping us from proceeding with the project.

2.5.5. Political Feasibility

All stakeholders are welcoming the proposed application, as it is a better, faster and easier to use alternative to any other recipe generator out there. We are not using any biometrical data, Social Security Numbers, business information or classified data, thus no sensitive data is being processed by our application. There is currently no law that interferes with the Hamamas project.

2.5.6. Economic Feasibility

The financial costs of this project are affordable. During development we have paid a \$5 monthly subscription from Spoonacular API of 5000 requests/month which was enough. To be able to deploy the app and make it accessible to everyone we will need to buy a domain name and get a cloud computing subscription. These costs are estimated at ~\$20/month which can increase based on the usage. In the beginning, an expense of \$25-30 per month is an affordable price for us three. In the future, if we will get investors, could be cooks or restaurants, we will easily have these costs covered. So for this overall cost we can conclude that the project is economically feasible.

Conclusion

Given the above technical, schedule, operational, legal, political and economic factors we can affirm that this project is feasible.

2.6. Risk Management

In order to properly handle the risks that can become impediments for our process and project, we decided to adopt a proactive risk management strategy. First we carried out a risk analysis to discover and get an insight on the potential problems that might occur. Moreover we established a risk mitigation, monitoring and management (RMMM) plan, in order to be able to anticipate and control the risks and know which actions should be taken to prevent and monitor, and maybe fix the problems if they happen. This way we made sure the risks affected our project to a minimal extent, if not at all. We have used a RMMM table to rank, prioritize and visualize our risks.

2.6.1. Risk Analysis

The risks are identified and categorized, then the occurrence probability(percentage) and the consequence level are assessed for each risk. The impact is calculated (probability*consequence) and this determines the importance rank that the risks are ordered by.

Risk ID	Risk	Category	Probability	Consequence*	Impact
1	Delayed deployment due to poor time estimations.	Business	60%	4	240
2	Late project finish due to overly high expectations for the MVP	Product	55%	4	220
3	Delayed workflow due to use of external dependencies	Technology	40%	4	160
4	Unseen bugs in the system.	Product	40%	3	120
5	Working with an unfamiliar technology/structure	Technology	33.3%	3	~99
6	Stakeholders lack communication interest.	Stakeholders	30%	3	90
7	Failure to follow chosen methodology.	Process	30%	2	60
8	Team members are unavailable and cannot work, could be because of health or personal matters.	Developers	15%	3	45
9	Team members interfering with each other's working tasks.	Process	20%	1	20

Risk analysis table

**Consequence is rated with a scale from 1 to 4, where 1 – insignificant, 2 – minor, 3 – moderate, 4 – major, 5 – catastrophic.*

2.6.2. RMMM Plan

After identifying and ranking the risks we made a plan for mitigating each risk so that we can minimize the probability of happening and the consequence if that risk occurs.

For each risk, a mitigation action is considered in order to reduce the probability and the consequence of risks (in case they occur). By monitoring we indicate which factors we should keep under surveillance to quickly discover when a risk occurs. Management describes the actions we should take in case the risk is not avoided and problems appear.

Risk ID	Risk	Mitigation	Monitoring	Management
1	Delayed deployment due to poor time estimations.	Make use of knowledge acquired from past projects and estimation formulas to accurately determine your timeframe.	The project is behind schedule, tasks are late.	Reorganize the workload to focus on high importance tasks, try to help each other and do more than planned for yourself if needed.
2	Late project finish due to overly high expectations for the MVP	Have realistic expectations for the MVP. Select the most important features to be included and leave the rest for a future version.	There is time wasted in “nice to have” features/design choices	Revisit the sprint backlog, cut the unnecessary parts
3	Delayed workflow due to use of external dependencies	Try to limit the number of external dependencies to a minimum. For the ones used, make sure they are quality assured, well maintained	There are bugs that you cannot figure out, because they happen on the external dependency	Write the respective functionality on your own, also let the dependency maintainers know about the problem
4	Unseen bugs in the system.	Test your code carefully before integration, follow the code standards 100%, ask for help you get stuck.	There are errors in the production application and unwanted behavior.	Find the bugs and fix them. Push the fixes as soon as possible to the live servers.
5	Working with an unfamiliar technology/structure.	Research on the technology you will be working with. Design and discuss before starting to code. If you have the knowledge, help your mates, if you are on the other side don't hesitate to ask for help.	The tasks take more than estimated. Developer gets stuck.	If you cannot find your way quickly, ask for help, so you don't waste a lot of time on aspects that may be easy. Use the web community to get around or your groupmates.
6	Stakeholders lack communication interest.	Make sure the you schedule the meetings ahead of time so that everyone is available	For different reasons, the meetings are canceled and the communication breaks. You don't know anymore if the stakeholders are satisfied with your progress and direction.	Have over-the-air meetings (telephone, skype, messengers, email). If a face-to-face meeting is mandatory, remind the stakeholder that canceling meetings can stop team from continuing with the project due to lack of feedback.
7	Failure to follow	Together with the team,	Team members fail to	The team together uses their

	chosen methodology.	have a refresh on how the methodology works.	follow the methodology and the schedule is endangered.	knowledge about Scrum to help each other properly follow the methodology.
8	Team members are unavailable and cannot work, could be because of health or personal matters.	Each team member has to maturely take care of himself/herself, and think about the wellbeing of the project/team.	Team member fails to attend meetings or work on his tasks. There is lack in communication and the schedule falls behind.	The available team members cover the sick one, and do more workload to compensate for their mate.
9	Team members interfering with each other's working tasks.	Always update the Scrum board on Trello and keep close communication within the team though Slack.	Team members are working on the same task.	Decide which member to continue with the interfered task and the other should pick a fresh one. Discuss, don't get upset!

RMMM table

3. Technical analysis

This chapter will deal with all the technical aspects of this project, and is split into two parts: software design/engineering and software development.

3.1. Software engineering

The following section will explain all about our chosen methodology, use cases, UML diagrams, prototyping, style guide, sprint documentation and code standards.

3.1.1. Code standards

In order to keep our code readable and easy to maintain and extend, we decided to agree upon a set of code standards that everybody has to follow. To make this task easier we chosen a Javascript library to help us, ESLint⁶.

ESLint is a flexible linting, code style utility for Javascript. It suites our project best because we use Javascript exclusively and it is highly configurable. We can define our own code "rules" (standards) as well as using an external set of rules, or both. The way we approached it is having a popular open source configuration as a starting point, that being the rules used at Airbnb⁷, and adding/overwriting that with our own preferences. The airbnb style guide contains a ton of useful rules, created and maintained by their engineers. It covers most of the standards that you can think of, from simple indentation spaces rules to not using certain syntax to keep your code readable. This made the choice easy, and gave us the feeling that we code like pros.

⁶ <https://eslint.org/>

⁷ <https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb>



```

1  module.exports = {
2    "extends": "airbnb-base",
3    "env": {
4      "node": true,
5    },
6    "rules": {
7      "prefer-template": "off",
8      "function-paren-newline": ["error", { "minItems": 5 }],
9      "consistent-return": "off",
10     "prefer-destructuring": "off",
11     "arrow-body-style": "off",
12     "no-unused-expressions": "off",
13     "no-underscore-dangle": "off",
14   }
15 };

```

In the picture above, you can see the configuration file for the backend project. To shortly explain it, we tell ESLint to use the “airbnb-base” config as a starting point, then define the environment to use “node”. The last key, “rules” contains the rules that we add/overwrite on top of airbnb’s.

Having this file and the plugin for our code editor of choice, VSCode, gives us live linting and error reporting, as well as pure code style warnings/errors. Using this tool made it a breeze for us to create and maintain our code standards guide, because we get the errors right in the editor, without having to check a rules file all the time.

One very big benefit for this approach is the linting/error checking functionality. Because Javascript is not a compiled language, you don’t have this features by default in editors, so ESLint brought us a lot of value in this regard.

3.1.2. Engineering methodology

One of the first subjects that crossed the table after we had found a good project, is how to manage the programming process. This is where our previous knowledge from the Software Design lectures (Software Engineering) came in handy. We wrote down pros and cons of the major branches in Software Engineering practises the so called Agile and Non-Agile methods and based on that we decided what was the best fit for our project.

3.1.2.1. Agile vs Non Agile

We knew that we should look into all of our options in order to find the best fitting solution for our team size, knowledge, project size, complexity and the development deadline. Before we started the Hamamas project we used our previous knowledge and made a list of positive and negative aspects concerning both an Agile and Non-Agile approach to project managing

⁸.

⁸ <https://www.upwork.com/hiring/development/agile-vs-waterfall/>

The positive aspects of non-agile methodology in our opinion are:

Pros of Non-agile

Clear deadlines: Its static nature and predictable workflow make it easy to create timelines, and stick to deadlines. This project had a clear deadline so this is one of the first things that came to mind.

Clear by design: Since each phase has a clear start point and a requirement review gate at the end of it, the team is forced to complete all tasks before the project as a whole can proceed. We knew that structuring the process would increase our chance to plan ahead with more accuracy.

Well documented: It requires documentation and a clear paper trail for each phase of development. This makes it easier to follow the logic of past projects and lay the groundwork for future projects. Following the steps and making sure everyone documents them would generate a good overview and bases for a solid report. Use cases and diagrams are for example a great way to define features and a good base for coding.

Easy learning curve: As the traditional approach to project management team members usually don't require any prior knowledge in order to start working on a project with this method. All of us have experience with this structure and are confident in using it.

Cons of Non-agile

Change can be time costly: The major downside to Non-agile is rigidity and the hampered ability to handle change. Testing occurs late in the project life cycle, and if you find out that your end users don't like the product you're building, it can be too late to pivot. We need to have a fast paced delivery in order to create our envisioned product, testing and meetings might increase or decrease the amount of features so that must be an option.

Slow delivery times: As many as four phases of development need to be completed before any coding begins—which means testers and customers won't see a working product until late in the life cycle. We want to show the product as soon as possible to see if there is anything we might have overlooked while thinking about the features.

Gathering requirements too early is risky: We did not know what we really want until we had a chance to play with a working prototype. Since non-agile handles all the requirement gathering upfront, there's a real risk of missing the mark or some cool features down the line.

The positive aspects of an Agile methodology in our opinion are:

Pros of Agile

Adaptability: The short development cycles give the project the flexibility to pivot when it needs to. In our case we wanted iterations off maximum one week and those would define how we would proceed. This was a huge need since our deadline was fairly short or such a complex application

Immediate user feedback: The emphasis on getting working products into the hands of our users meant that the project is guided by the market. This reduces the risk of building an app that nobody wants while increasing the chances we will find that killer feature that will 'sell' our product earlier in the project life cycle.

Test-driven development (TDD): The beauty of breaking a project into manageable chunks is that there is enough time to write unit tests or even do some form of user testing for the few features that made the cut for the MVP.

Fast, high-quality delivery: TDD at each iteration leads to fewer bugs and higher-quality releases. A solid foundation leads to quicker, higher-quality releases with successive iterations.

Teamwork: Agile methodologies place an emphasis on frequent communication and face-to-face interactions. Our team worked together in sessions, we benefited a lot from pair programming, and spoke daily via slack.

Cons of Agile

Non clear timelines: With all of its advantages, Agile's flexibility can also easily leave the door open to procrastination, leading to postponing meetings or adjusting timings a moment in advance. Since tasks are often being reprioritized or added with every iteration, the overall timeline can seem to stretch into infinity.

Skill-dependant teams: Agile is designed for a small multidisciplinary team. This often means only one person per role. The relative lack of structure when compared with Non-agile means that each member must be self-disciplined and proficient in their role. In our group this meant that we had a lot of questions back and forth to help all group members contribute to the project.

Time intensive: Agile works best when everyone is committed to the project. This is especially true for Agile because much of the methodology is focused on active team involvement and face-to-face collaboration, which can be more time-consuming than the traditional approach.

Conclusion

Based on the requirements we have received for the project (timeline, size and complexity) and the size and overall knowledge of the team we have decided to go for an Agile methodology. The reasoning behind it was based on the following:

- We wanted to create a MVP fast and test is with actual users
- We did not receive any specific requirements, we made them ourselves
- Not every team member is proficient in the same programming languages/frameworks
- We want to be able to pivot when we/our user tester believe to have found a good feature
- We have to be able to finish the application within the deadline with a small team of three people

Assessing this short list we saw that we needed adaptability, fast and test driven delivery and most of all teamwork in order to keep the team up to speed. Predefined structure and documenting is important, but it could not outweigh our need to work with our users and pivot when we saw fit. That being said, we have used Use Cases in order to define our features so some of the non-agile practises have been used to our advantage.

All of this reasoning led us to believe the Agile approach would be the best fit for our project.

In the next paragraph we will go more in depth on which Agile methodology we have chosen, why we have chosen it and how we have implemented it for our project.

3.1.2.2. XP & Scrum

As you have read in the previous paragraph we decided to implement an Agile methodology, that being said we still had to decide which one or a combination of which methods we will be using for the project. In order to find out the best solution we used the following criteria to determine our most suitable method.

- We need to be able to constantly work on the product.
- There is a knowledge gap in the group and this has to be handled
- Everyone has to be constantly aware what is being done, how and why
- We need to set short deadlines, cut off points
- We want continuous delivery so outsiders can test it
- We want to deliver a working MVP ahead of the deadline to make more improvements

Pair programming and sprints

Using this shortlist we looked into XP and Scrum. One of the main issues was a knowledge gap in the project. We believed that pair programming would solve that. Besides that we wanted to implement test driven development. So we decided to implement a base of eXtreme Programming. Besides that we wanted short iterations so we can pivot whenever we come up with a feature or fix, sprints were chosen for that since we had experience with them.

Scrum Board and DoD

We used the exploration phase in order to determine our user stories and divide up tasks, by doing so we managed to make a backlog of tasks. Also we decided that the team has collective code ownership, anyone was allowed to review code changes. A Scrum board was created using Trello, and we implemented test driven development by using pull requests on GitHub (every time anyone wanted to merge their feature or improvement, the rest of the team will look into it first and test it). During our first sprint we defined our definition of done as follows:

- The feature must meet all the acceptance criteria
- The code must be up to standard with the code standards
- The feature must be able to be tested stand alone and merged

Sprints and prioritization

After every sprint we had a retrospective and we planned the next sprint accordingly with the help of our backlog. Since we assigned roles in our team, the product owner prioritized task that went into the sprint. Since we had no clear indication of time when a feature would be made, we did not implement burndown charts also we did not have a daily scrum, but instead had daily communication through slack. We did not plan any code freeze because we were a small team and well aware of which branches would be merged.

Conclusion

To sum it all up we will be using common practises from XP and the sprint structure from SCRUM.

- Non-agile
 - Use cases - used to reinforce our development process
 - Use case diagrams - used to visualize actors and their rights
 - Sequence Diagram - used to visualize flow of objects
 - System Sequence Diagrams - used to visualize flow of system and actors.
- Agile
 - XP
 - Pair programming - used to level the knowledge within the team
 - TTD - used to ensure every feature branch is working from the start
 - Collective code ownership - create a sense of responsibility within the team.
 - Small releases - used to ensure full functionality of the application
 - Coding standard - ensure all code is readable and understandable by the whole team.
 - Scrum
 - Sprints - Ensure structure and give scope to iterations
 - Sprint review and retrospective - Ensure constant growth and learning from experience and mistakes
 - DoD - Ensured that every member knew exactly what it meant for a task to be complete.
 - Roles - Gave a sense of assurance to be able to ask or rely on a group member that had a role assigned
 - Product owner - Made sure that features were well documented put into the sprint

By combining mostly XP and Scrum we had a stable and agile framework to work with. We selected the necessary programming practises to our advantage and it worked well for this relatively small project. We do believe that the size of the group played a huge role in its ability to be flexible with using different aspects of agile development. If the group had been larger or the project required more time, we would have chosen to use more structure in order to make the team more streamlined.

3.1.3. Team structure and roles

Even though our team was not big we did appoint certain responsibilities to each other in order to have a better workflow and making sure we use the strengths of each team member to the fullest.

- We decided that Dana would act as a product owner, she will prioritize tasks from the backlog and she would have a final say in whether a feature would be implemented or not.
- Petru was made in charge of release management, making sure that all the merges were done correctly and met the DoD.
- Mac was in charge of user testing and managing the project process making sure the sprints were documented and everyone knew who was working on what.

By doing this we always had a good fallback when decisions had to be made and we knew who to count on when anything arose.

3.1.4. Use Cases

We wrote some brief use cases and fully-dressed ones as functional requirements, to help us determine how a user can achieve a specific goal. We made use cases for sign up, login, browse generated recipes, delete account, change password, mark recipe as favorite, browse favorite recipes, browse recently viewed recipes and logout.⁹

UC 1: Browse Generated Recipes - Fully Dressed

Scope: Hamamas Recipe Generator, View Generated Recipes functionality

Level: User goal

Primary Actor: User

Stakeholders and interests:

- User: wants to see recipes based on selected ingredients and filters in a clean and easy manner.

Preconditions:

- User has logged into their account.
- User is on search view.

Postconditions:

- User can view recipes

Main Success Scenario:

1. The user types in ingredients that they wish to use for the dish.
2. The user chooses the option “Feed me”.
3. The system prompts an image with recipe title.
4. The user can now swipe left for viewing a new recipe, or swipe right to go to the instructions of the displayed recipe.

Alternative Flow:

1. If the system fails: make sure there is internet connection.
2. If the user hasn't selected any ingredients:
 - a. The system shows an error with appropriate and informative message.
 - b. The user types in ingredients.
 - c. The user chooses the option “Find Recipes”.
3. If the user inputs cannot find ingredient:
 - a. Make sure the spelling is correct.
 - b. Choose another ingredient.
 - c. Search for recipes again.

Special Requirements:

- A device able to connect to the internet through a browser.

Frequency of Occurrence: Every time the user want to view recipes.

⁹ More Use Cases can be viewed in the Appendix section.

UC 2: Filter results - Fully Dressed¹⁰

Scope: Hamamas Recipe Generator, Filter functionality

Level: User goal

Primary Actor: User

Stakeholders and interests:

- User: wants to filter recipes based on selected options

Preconditions:

- User has logged into their account.
- User knows how to navigate in the application

Postconditions:

- User can always adjust filters
- User can search for recipes with filters applied

Main Success Scenario:

1. The user navigates with a swipe motion to the filters page
2. The user selects filters (optional).
 - a. Either sets a filter on or off or uses a slider.
3. The user chooses filters that she/he wants to apply.
4. The user can now swipe back to search page and use search field.

Alternative Flow:

1. If the system fails: make sure there is internet connection.
2. If the user selected too many filters:
 - a. The user can go back to the filters page and redo his/her choice.
 - b. The user restarts the app.
 - i. sets filters again
3. If the user inputs cannot find ingredient:
 - a. Make sure the spelling is correct.
 - b. Choose another ingredient.
 - c. Search for recipes again.

Special Requirements:

- A device able to connect to the internet.

Frequency of Occurrence: Every time the user wants to filter.

UC 3: Delete Account - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.
2. The system displays the home page.
3. The user selects to view their profile.
4. The system displays the user profile.
5. The user chooses the option to delete their account.
6. The system prompts a validation dialog asking if they are sure to delete the account.
7. The user approves deleting the account.
8. The user account is now deleted and the user is redirected to the landing page.

¹⁰ We thought about this use case and wrote it down, but this one is not included in the first release. We did not want to remove it though, as it will be useful in the future.

UC 4: Change Password - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.
2. The system displays the home page.
3. The user selects to view their profile.
4. The system displays the user profile.
5. The user chooses the option to change their password.
6. The system prompts a dialog where user has to input new password.
7. The user types in required information.
8. The user approves changing password.
9. The password is now updated and user is redirected to their profile.

UC 5: Mark a recipe as favorite - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.
2. The system displays the search page.
3. The user inputs search terms and chooses the 'Feed me' option.
4. The user gets results.
5. The user swipes left until he can find the desired recipes, then swipes right. (Buttons can be used as well instead of swiping.)
6. System displays instruction page.
7. The user can choose to favorite the recipe using the top bar.
8. The recipe is now saved in the favorites list.

UC 6: Mark a recipe as made - Brief

Primary Actor: User

Main Success Scenario:

1. Repeat steps 1-6 from "Mark a recipe as favorite" use case.
2. The user chooses the "I made it!" option.
3. The recipe is now saved in the user's database as made.

UC 7: Browse favorite recipes - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.
2. The system displays the home page.
3. The user selects to view the favorite recipes from the bottom navigation marked as a star icon.
4. The user gets a list of recipes that he/she marked as favorite before.
5. The user can now choose one of the recipes and display its cooking information.

UC 8: Browse recently viewed recipes - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.

2. The system displays the home page.
3. The user selects to view the recently viewed recipes/the history from the bottom navigation marked as a clock icon.
4. The user gets a list of recipes that he/she viewed before.
5. The user can now choose one of the recipes and display its cooking information.

3.1.5. Prototypes

In today's world everything that's more than a few clicks away is not good enough, and that is the reason why we put great effort into user experience for this software. It is common sense that a user will always choose a website that is easy to use and navigate through, with appropriate feedback for his/her actions, and a clean interface over one with cluttered screen and difficulty to find the desired information, but it can be pretty easy to go down that path. By adding fancy content, shiny buttons and all kind of imagery in hope of being "creative" and having website which is not "boring", it can happen that the user becomes distracted from the task they are trying to carry out and becomes confused about the point of the application/specific page (view) of the application. In this case not only the user will leave the application, but as shown in recent research users experience feelings of guilt when they fail to use an interface¹¹ (it would be the same as failing to use a printer or an overly complicated coffee machine).

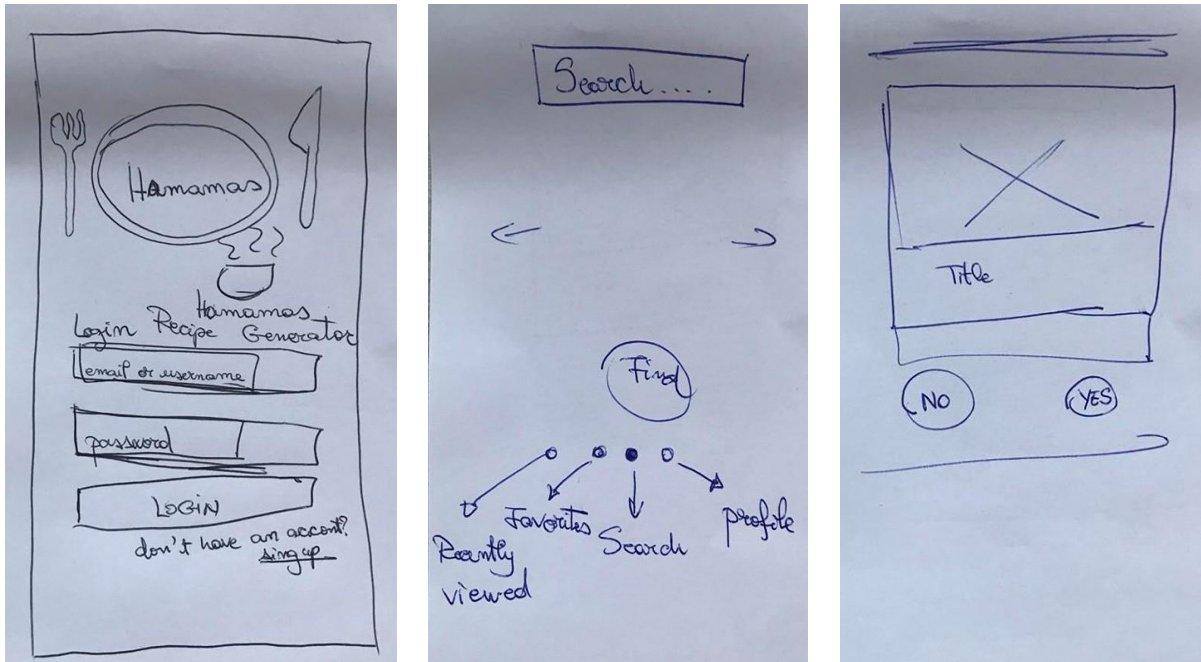
So we wanted to keep things nice and simple, but fast, interactive and enjoyable. We liked the swipe feature for the list of generated recipes as it is a fine alternative to process information quickly compared to the never-ending classic vertical lists. Also, swiping is now considered a "cultural phenomenon"¹² since a great deal of successful companies are using it. That is why we went for a minimalistic design and a Tinder-like swipe view for the results recipes.

3.1.5.1 Paper Prototype

To achieve the interface that we wanted and a good user experience we first started out with designing some low-fidelity prototypes, namely paper prototypes. This was the first step of materializing our idea into something visual and as the saying goes "an image is worth a thousand words". That was also our case, since we made each other understand our viewpoints better and we got a basic idea of how the application will look. We sketched out the login/sign up, search view, results view and favorites.

¹¹ <https://www.uxpin.com/studio/blog/bad-ux-makes-users-blame/>

¹² <https://iq.intel.com/how-swipe-left-swipe-right-became-a-cultural-phenomenon/>



Paper prototypes for the login screen (with 2 different logo ideas), search screen and results.

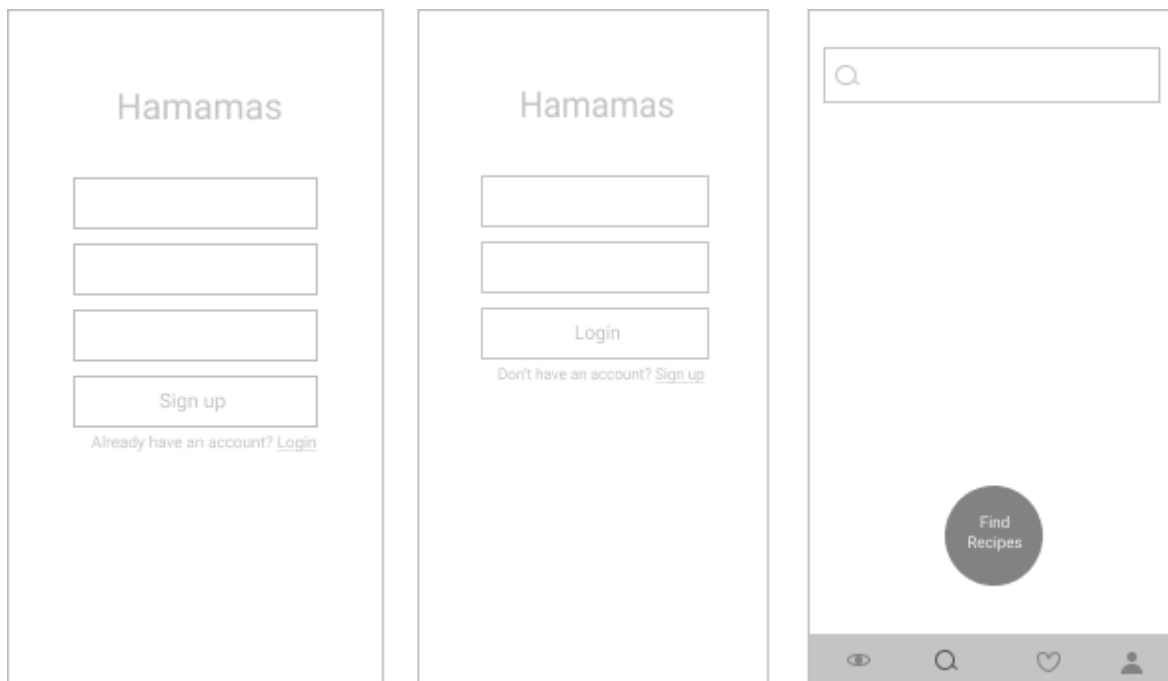
3.1.5.2. Wireframes

Next step was to create some wireframes, but by this time we had already spotted that there were some design flaws in our prototyping. For example, at the beginning we had this idea of swiping between views (search and profile), but after we “played” a bit with it and imagine how that would feel for the user, we found out that it can be pretty confusing to use the swipe for both the results and the navigation. Then we made a new prototype with a traditional bottom navigation with meaningful icons for routes, as we considered it makes more sense for the user to know where they are in the application, and what other options are available.

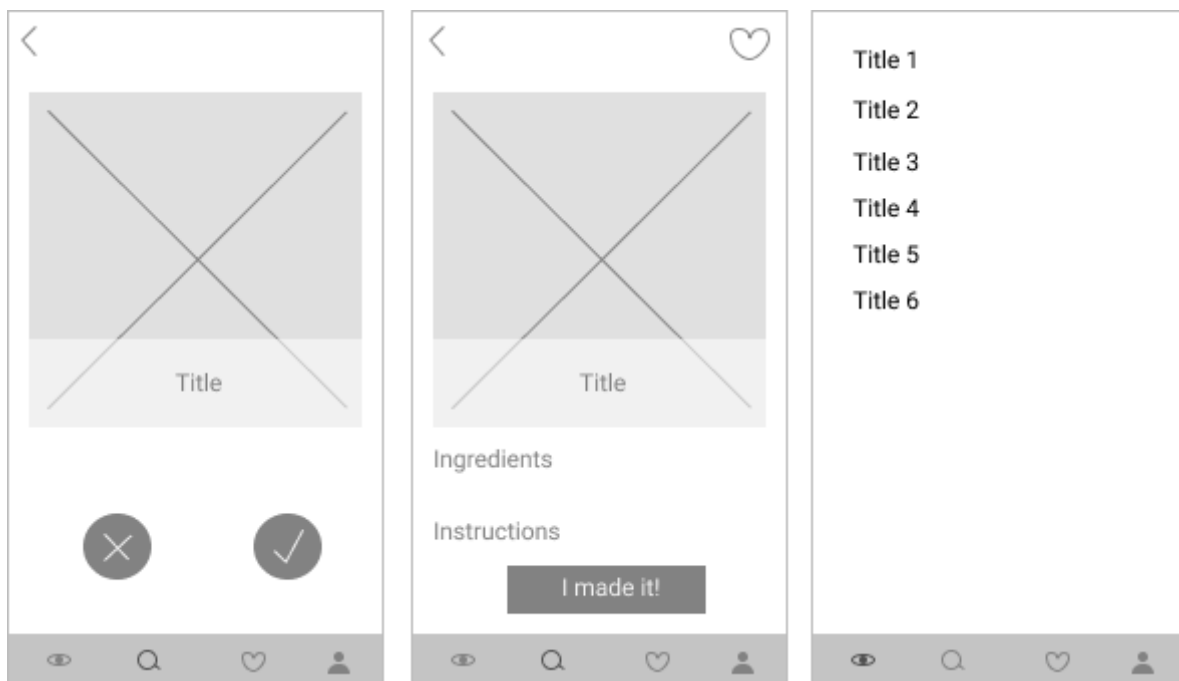
Another example would be the favorite recipes: first we wanted that to be a part of the profile view, so the user clicks on a link which is found on the profile view and he/she will be redirected to the favorites view, when in fact it makes much more sense to add that in the bottom navigation for quicker accessibility. We created some wireframes based on these premises and our previous paper prototypes, which were also tweaked with small changes several times after doing some testing on our friends and getting some feedback from them.

During this phase we agreed on the information architecture (what should be put where) and navigation between views.

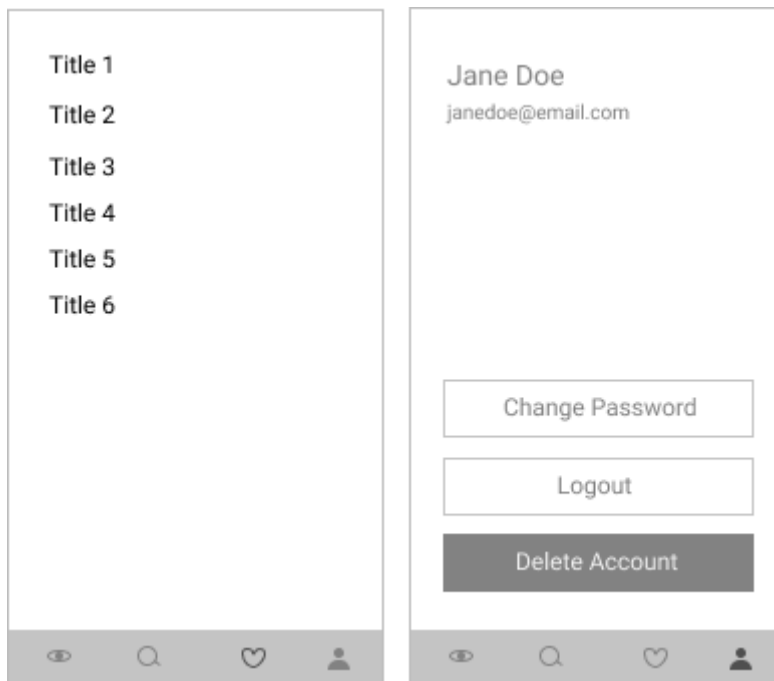
The wireframes displayed below represented the skeleton of our user interface.



Wireframes for sign up, login and search views



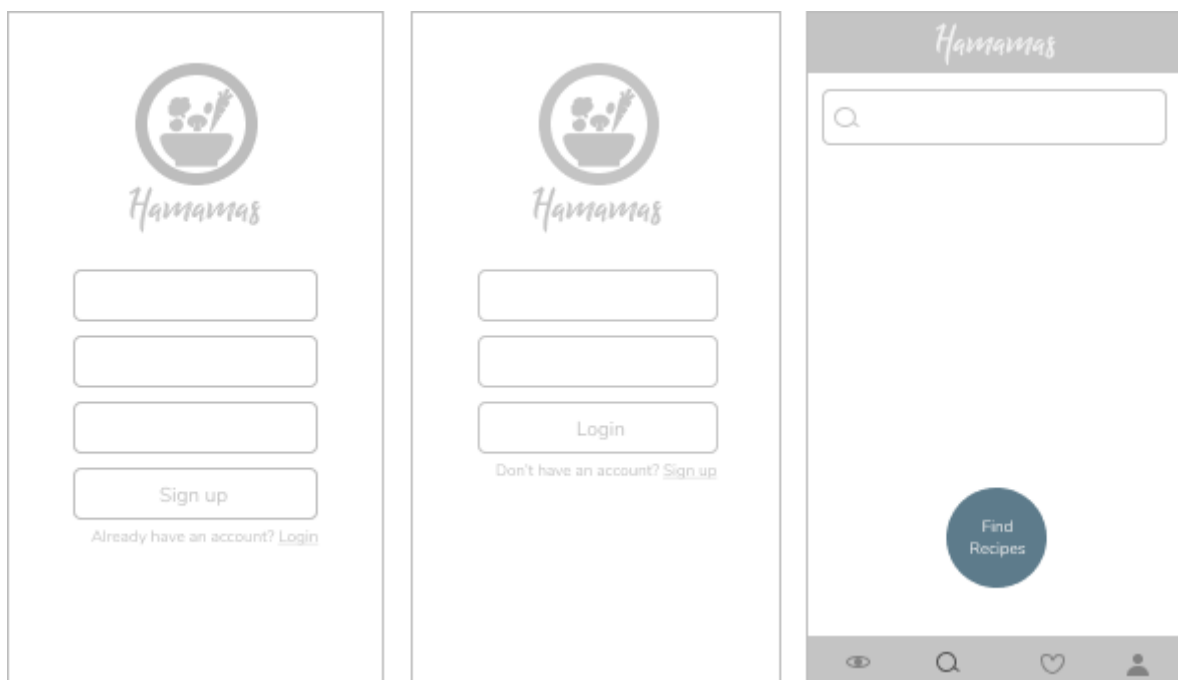
Wireframes for results, cooking instructions and recently viewed views



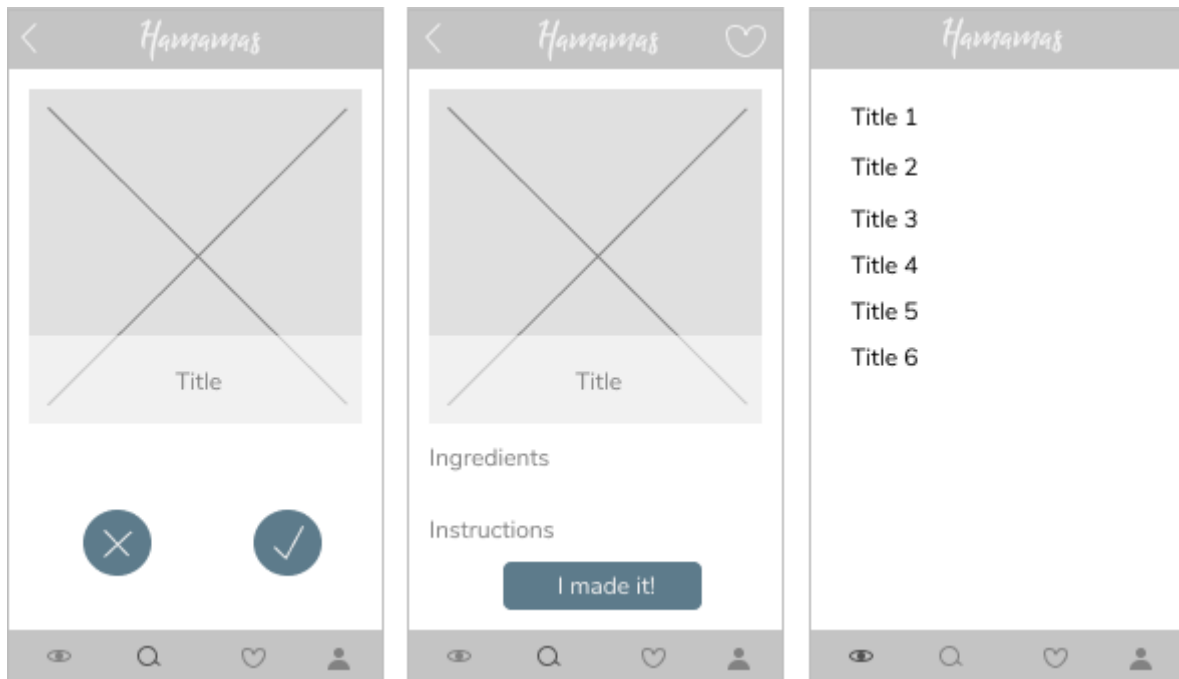
Wireframes for favorites and profiles views

3.1.5.3. Mock ups

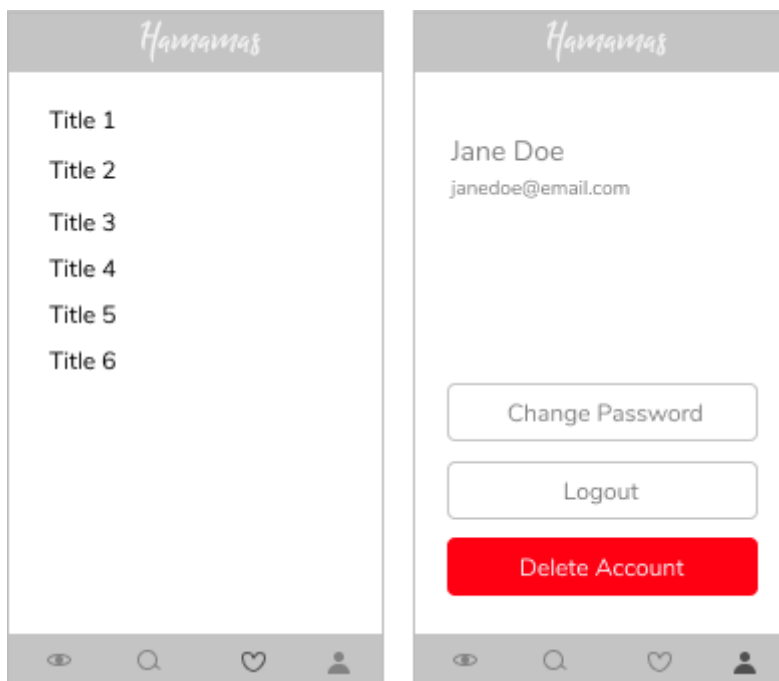
The last step in our prototyping was the mock ups. Since we were really happy with the wireframes we made, in this stage we only added the text font, colors and logo. We also added the top bar to all views (except login and sign up).



Mock ups for sign up, login and search views



Mock ups for results, cooking instructions and recently viewed views



Mock ups for favorites and profile views

The whole prototyping and designing phase was very beneficial in so many ways:

1. we realized that we are on the right track, this is what we and our users want, UI and functionality-wise.
2. we got a clear idea of the UI.
3. we already got some feedback from potential users, which lead to improvement of the UI and UX.
4. we saved time and effort when developing because we already knew how everything should be arranged in the viewport and styled.

5. it helped us take some software architecture decisions, more specifically this way we figured out that most of our views don't share state, so we agreed on having each view component managing its own state rather than having the most top level component doing that.

3.1.6. User stories

Based on our use case and prototypes the Product Owner created user stories, which helped in planning the project.

1. As a user I want to sign up with username, email and password.
2. As a user I want to login with username and password.
3. As a user I want to be able to change my password.
4. As a user I want to be able to mark recipes as favorite.
5. As a user I want to view my favorite recipes.
6. As a user I want to search for recipes by inputting ingredients.
7. As a user I want to look through the results of the recipe search.
8. As a user I want to view my profile.
9. As a user I want to view session history.

3.1.7. Style guide

As mentioned before, the UI is simple enough so that it's not cluttered, but fancy enough so that it's not boring, due to both our personal preferences and considerations such as: a mobile screen does not provide a lot of space, and the nature of the application (recipe generator) means there will be displayed a lot of information of different types the user has to process (pictures, ingredients, instructions, many numbers, buttons, etc).

Logo

The logo is our creation and it backs up the fact that Hamamas is a recipe generator application. It shows our commitment, it helps us stand out and be different from competitors, and it gives our brand memorability as people are more likely to remember a logo than the name of the company/software product.



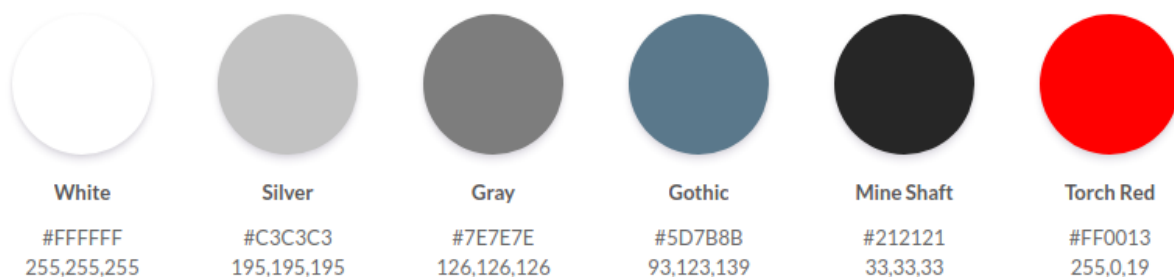
Hamamas Logo created using Figma

Colors

The color scheme features white, blue, red and different shades of grey.

Usage and meaning:

- White - we used it for top bar and bottom navigation bar, and it is associated with brilliance, cleanliness and perfection
- Silver - used as background color, since a white background would be too bright and blinding to the human eye; it is a formal and sophisticated color
- Gray - used for text; it reminds of the strength of the black, but it's not as powerful and aggressive as a plain black
- Gothic - used for some of the buttons; it is a light and friendly shade of blue
- Mine Shaft - used for focused icons in the navigation bar; it communicates the authority of black and it contrasts well with the silver color (for example, if a user is on the search view, the search icon is mine shaft and the other icons in the navigation bar are silver)
- Torch Red - used for the delete button, signifying danger



The colors we used with name, HEX and RGB codes

Typography

We used two fonts for this applications:

- Billy Ohio in weight 400 - for 'Hamamas' title on top bar, login and sign up views
- Nunito in weight 300 - for paragraphs and weight 600 for titles

Aa

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890(.,:;!\$%*)

Billy Ohio Regular
Weight:400
Style:normal

Aa

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890(,.;?!\$&*)

Nunito
 Weight:300
 Style:normal

Aa

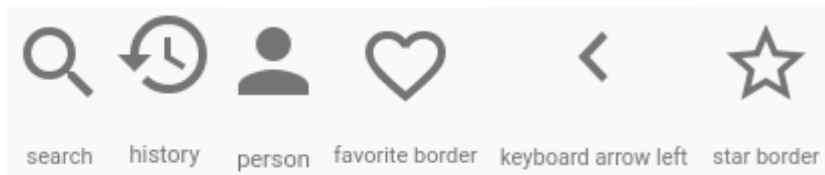
ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890(,.;?!\$&*)

Nunito
 Weight:600
 Style:normal

Examples of Billy Ohio and Nunito typefaces

Iconography

We used Material UI and Font Awesome icons for the top and bottom bars. Icons are meant to summarize some content, thus they are very useful when designing for a small device such as mobile. The ones we used are quite popular and intuitive, so users (most of them) are not in doubt of their meaning. Including icons in an application also helps achieve a better looking UI.



The icons that we used with title

3.1.8. Minimum viable product

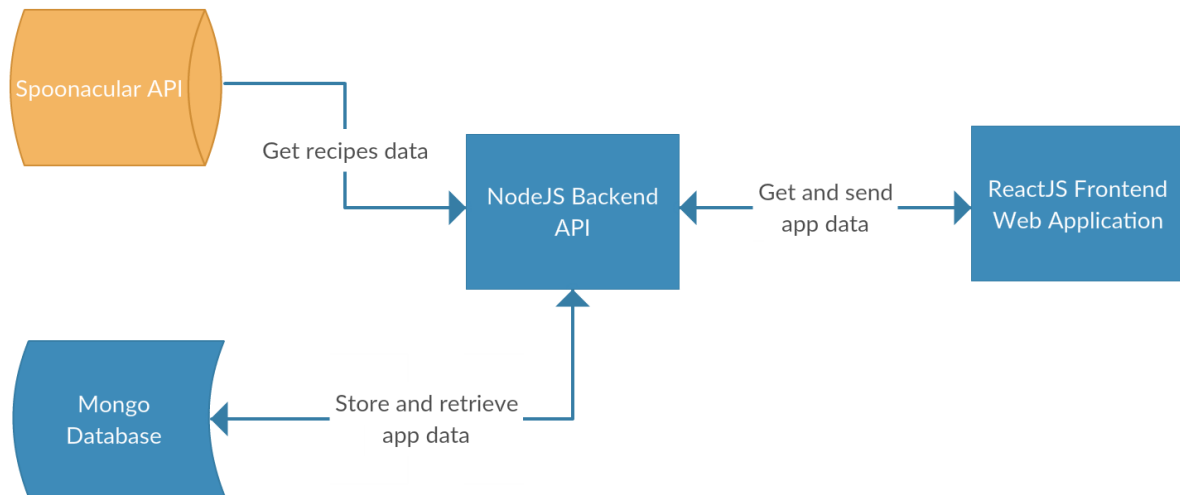
For our project the Minimum viable product a.k.a. MVP was really important. We wanted to test our application during its production to get feedback from outsiders that might have improvements that we did not think of. We already saw the huge benefit of showing prototypes so we wanted that to continue throughout the development process. In order to have a MVP we needed to define its scope as a team we had a meeting and decided that the following features needed to be implemented for the MVP to be complete:

- Register
- Login/Log out
- Search (using the API)
- Results (using swipe gesture and API)
- Favorite a result
- See overview of favorites
- See your history

When these requirements were met we gave it to outsiders to test. During development each feature was tested separately and as a whole with the rest of the application.

3.1.9 UML

3.1.9.1. Architecture (overview)



App Data Flow Diagram

The architectural design of our application is quite simple

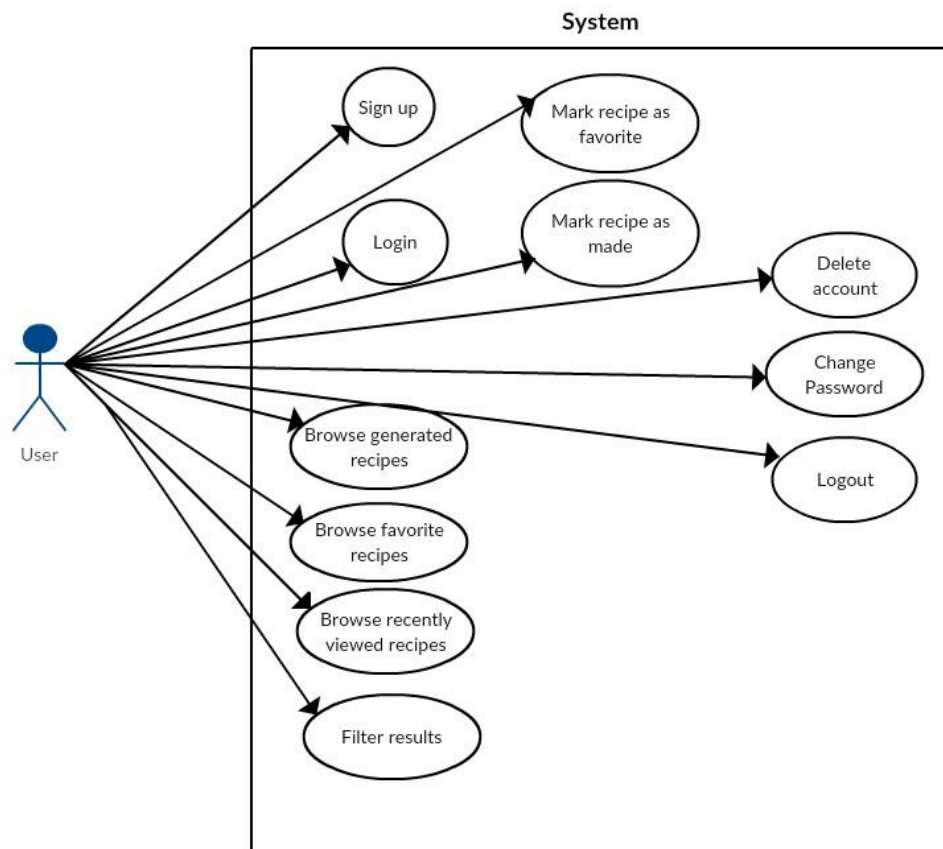
- NodeJS Backend API (Server)
 - the central unit of the app, all the app data goes through it
 - handles requests from the frontend app
 - stores and retrieves data from the database
 - gets recipe data from Spoonacular API¹³
- Mongo Database
 - stores our local data (e.g. user information)
- Spoonacular API
 - used to get recipes data to feed our app
- ReactJS Frontend Web Application (Client)
 - frontend entity, what our end users see and interact with
 - displays the data retrieved from the Node API
 - sends data to the Node API to be stored in the database

Using this approach we separate concerns into individual pieces that can be independently updated, improved etc. One interesting choice here is the separation between the frontend web app and the Node API. Another way to do it would have been to integrate it all into a full node application that would serve dynamic HTML files based on the client requests. This would have been faster to develop but a little messy and not **future proof**. By doing it this way, having the backend totally separated from the frontend, we were allowed to use the power of an awesome framework like React. Not forget to mention, if we will develop the native mobile app we can use the backend with no modifications, it will work the same for whatever frontend approach we will take and this will be a big benefit.

¹³ <https://spoonacular.com/food-api/>

3.1.9.2. UCD

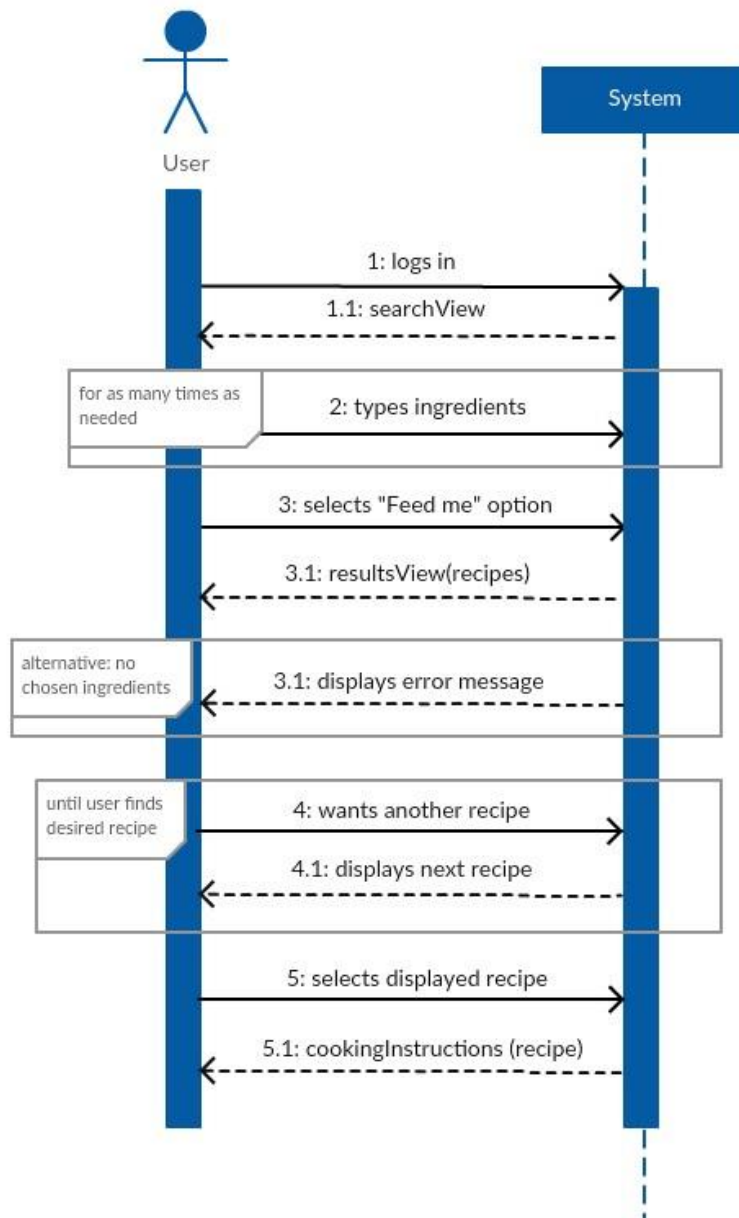
Based on all our use cases we made the below UCD which expresses all the functional requirements of our system. All use cases except “Sign up” and “Login” require the user to be logged in in order to be performed.



Use case diagram for Hamamas System

3.1.9.3. SSD

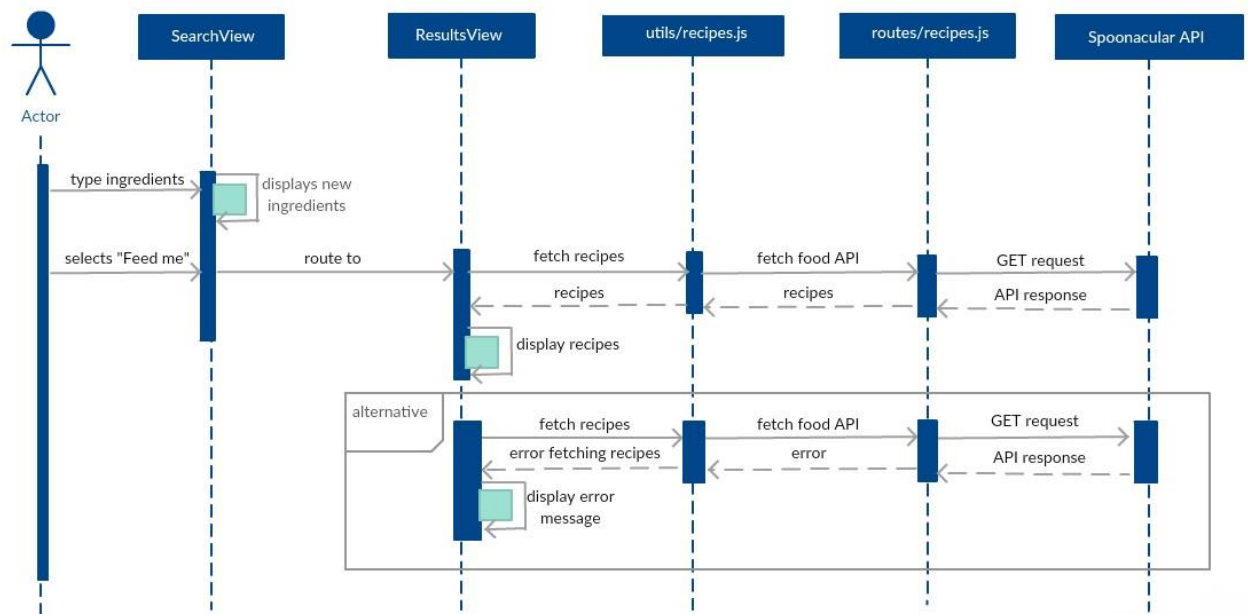
Based on the “Browse generated recipes” we created the below system sequence diagram, which captures the most important functionality which is how we get the results recipes. There can be visible the events that the actor (user) produces, the returning messages or events from the system, and loops.



SSD for browsing generated recipes scenario. We identified two loops: the first one regarding typing ingredients and the second one about viewing the next recipe, if the user does not want to see the cooking instructions (does not want to select) the currently displayed recipe. As an alternative scenario, the user might choose “Feed me” before inputting any ingredients and in this case the system will show an error message informing the user about the problem instead of displaying the results view.

3.1.9.4. SD

Even though we don’t have classes per say, we are using React components and JavaScript files containing related methods, so we could create a sequence diagram illustrating the interaction between our “classes” over time in the case of choosing ingredients and then fetching recipes based on those ingredients from the API.



SD for fetching recipes. The event starts with the user typing some ingredients using the UI, namely the search view, which re-renders displaying the newly chosen ingredients. Then the user selects the “Feed me” option, which will cause redirecting to the results view (another UI component). Here the ingredients that have been received will be sent to an util method which fetches the food API. A GET request will be made to this API and based on the response the results view will either display the recipes or an error message.

3.1.9.5. Database model

We needed a database to store the users and information about each one of them such as username, email, password, favorite recipes and made recipes, so we considered this would be a good use case for MongoDB instead of an SQL database. We have a simple user schema created with Mongoose¹⁴.

```

const UserSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: [4, 'The username should have at least {MINLENGTH} characters'],
    trim: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
    trim: true,
  },
  email: {
    type: String,
    trim: true,
    unique: true,
    sparse: true,
  },
  name: String,
  favorites: [{
    id: Number,
    title: String,
  }],
  madeRecipes: Array,
});
  
```

User schema for Hamamas application

¹⁴ <http://mongoosejs.com/>

3.1.10. React Components

By this point, with the help of the use cases, diagrams and both the low- and high-fidelity prototypes we created, we had a basic idea about what our architecture will look like. On top of that we wanted to decide on how to structure the frontend, so we tried to have a React component-like thinking and sketch out the so-called component tree. We basically have three kinds of components: *App.js*, “container” components and dumb components. We chose to call the last two categories like this since it seemed easier and understandable for all of us, however in the React development world they could also be referred to as “*Fat and Skinny, Smart and Dumb, Stateful and Pure, Screens and Components*”¹⁵ or simply “*Container and Presentational components*” (this is the terminology Dan Abramov, creator of Redux, is using). Regardless, we will describe each of them in more detail in the following paragraphs.

On the top level we have the *App.js* as first component that is rendered. This component does not contain state, but it deals with routing and includes all the “container” components. This is where we define the routes for each view, and this component also decides what to render based on whether the user is logged in or not.

```
{/* Add the routes here */}
<Route path='/search' component={Search} />
<Route path='/profile' component={Profile} />
<Route path='/results' component={Results} />
<Route path='/instructions/:recipeId' component={CookingInstructions} />
<Route path='/recent' component={Recent} />
<Route path='/favorite' component={Favorite} />
<Redirect to='/search' />
```

Routes corresponding to each “container” component in *App.js*

Next we have “container” components, describing how things work. They are complex, stateful and deal with different functionality. They display content and render other components, and call various util methods that help with fetching user data, updating information about the user and fetching the recipes from the food API.

```
└─ containers
   JS CookingInstructions.js
   JS Favorite.js
   JS Login.js
   JS Profile.js
   JS Recent.js
   JS Register.js
   JS Results.js
   JS Search.js
```

All the “containers” we will have

The last type is dumb components. These are children of the above mentioned components, do not have state, and they are *styled-components* i.e. they describe styling and/or deal with

¹⁵ https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

displaying data. They are reusable, for example based on the mock ups that we had we could see that the same *Recipe* component will be used in both *Results* and *CookingInstructions*, but we just pass down different data to it. We included here the bottom navigation and the top bar. These were some dumb components that we assumed we would have for sure. We didn't add more here before coding, because we agreed that if someone is working on a main feature (which is a view or a "container" component), they will break down the mock up into smaller components in the way they see fit.

Although this was a minimum analysis of which elements we will have in our architecture, it was very useful because this way we made sure we understood each other's vision, so we all knew how the code should be structured and follow the same pattern. To be more specific, we agreed that whenever we wanted to create a new main feature we do the following: create a file in "*containers*" folder which will have state and functionality, import it in *App.js* and define a route for it, then add its children to either the file itself (not in the component class) or to a "*components*" folder (which is only for stateless dumb components).

Among the benefits of splitting our components like this were: a better separation of concerns, better reusability and simpler components.

3.1.11. Sprint documentation

In this chapter we show the overview of our approach and sprint documentation. Due to the size of our team we did not have any code freeze period. We had a continuous delivery cycle that made our project fast and agile. In order to have a consistent workflow we decided to work with sprints of 7 days. Each sprint consists of planning, executing and review + retrospective. The whole team attend the planning, review and retrospective meetings. During the sprint we had meetings whenever needed. As you can read we combined review and retrospective, this is done by choice due to the fact that the team was so small.

Implementation of Sprints

Each sprint consists of planning, execution, reviewing and the retrospect as aforementioned. Before we dive into each individual sprint we would like to address the overall process that we ended up using because it does not conform to the usual sprints that are being conducted when using pure SCRUM.

No daily scrum

Due to the size of our team and distance of traveling we decided to update each other by using Slack. The team has gathered on many occasions to pair program or to discuss features and improvements, but this did not happen on a daily basis. Since we were only with the three of us we did not need a constant update on who is working on what because we had this overview via Slack and GitHub. Besides writing each other on Slack we divided the tasks when we met up, created feature branches and knew exactly who was responsible for which branch.

No code freeze

We used a 7 day sprint cycle, it is quite long for a XP iteration but we still adapted TDD from XP in every sprint. With our 7 day cycle we ended with reviewing and retrospecting and deploying into our master branch. The reason why we had no code freeze is because our code changes were not impacting each other, we used a component based library so features did not depend on each other. A code freeze makes room for testing and making sure no regression bugs have slipped into the code, since we did not have huge interconnecting features and we tested everything ourselves first we saw no need to code freeze and postpone our deployment.

No Scrum Master

The team members did get roles assigned but we all agreed that everyone is responsible for maintaining the Scrum Board and everyone was allowed to add features (user stories) to the backlog but only the PO (Dana) was allowed to assign tasks to a sprint. The team was just too small to need an extra form of management.

#1 Sprint - 5th - 11th November

The first sprint is always slightly different from others due to its nature to harness structure and to create a strong foundation that can support the project throughout its duration. This being said we also needed to refine the project idea, make it more visual so the whole team can have a good overview of what is desired and what should not be added (or at least not in the beginning). Coming back to the structure, a definition of done (DoD) and acceptance criteria were also being mentioned this was needed to ensure that when the development would actually start everyone knew exactly what the standards are and how they needed to develop in order to meet the level of the team.

Sprint planning

This was the initial sprint, its main focus being the setup of the project and third party solutions to help us get started. We met as a team and bundled our combined experience to setup the most effective environment possible that suited everyone in the team.

Sprint execution

Without going too much into detail we did the following tasks

- We talked about features - What to add or not
- We made use cases - Based on the chosen features
- We made user stories - Based on the created use cases
- We setup our Git - Needed a repository and version control
- DoD - definition of done was needed for transparency
- Set up Trello - Digital Scrum board
- Set up Slack - Communication medium
- Chose IDE - Programming environment
- ESLint added - code standard assistant

We added favorites and history use cases to the backlog. In order to get the MVP done in time we want to prioritize the following use cases for the next sprint:

- Register

- Login

Review & retrospect

We decided to prioritize Slack over any other communications channel. All the third party solutions are well in place (Trello, Slack, Github) and working since we decided to use code standards and ESLint to help us with upholding it we needed to set it up in our IDE as well. We needed to adjust ESLint for Windows, due to the `/r/n` (new line) rule which is different for Windows and iOS. We had some issues there but solved them eventually.

#2 Sprint - 12th 18th November

With the foundation well set up we now wanted to focus on the visualization of the application. The team met up on several occasion to discuss features, UX possibilities and color schemes. Besides the look and feel we needed to visualize the actual interaction and flow of the application by using a variety of diagrams. On top of that we wanted to create a draft model of our Database according to our current assumptions. Now that the inception part was past us we wanted to create a backend that would support our feature development in the long run. Beginning with the backend was challenging but as we progressed we saw the benefit of putting up a good code base to start from. It was also a good way to test our code standards and our ESLint implementation.

Sprint planning

Main focus of the sprint was modeling the application before we can start to develop it and make a start with the architecture of the application. Mockups and prototypes were the first priority.

Sprint execution

After creating use cases and prototypes, we worked on two task that were prioritized as fundamental for the project. The team will pair program these task so everyone will have a proper understanding of the structure.

- We made prototypes - first form of visual aid
- Made mockup - made based on the chosen prototype
- Made a use case diagram - based on use cases
- Made System Sequence Diagram - Based on desired interaction
- Made Sequence Diagram - Based on desired flow
- We made the user model for our MongoDB DataBase
- Started with backend development
- Features that were on planned for the sprint
 - register
 - login

When features were deemed ready we reviewed the pull requests as a team.

Reviewed pull request for

- login
- register

Based on the prototypes we created user stories and by using those we added additional use cases to the backlog.

Added the following use cases to the backlog:

- SearchView
- Results Swipe
- Instructions
- Topbar
- Routing

Review & retrospect

Adding backend code was a good start just to have a base to work from as soon as we start the actual full development process. We decided to use class components just to keep a neat coding standard, functional components could be useful, but do not any extra benefits so it's better to stick to our standard.

#3 Sprint - 19th 25th November

With the backend in place and the prototypes done and diagrams created we could focus on the feature development. We each picked two features to work on in this sprint. This sprint showed the power of React, since one feature added components that others could use or easily change and reuse. This meant that the more tasks were finished the easier it became to pick up the next one and finishing it up before the end of the sprint. Being able to refer back to the mockups and diagrams gave us all the information we needed to develop all the way up to the review and retrospect. Besides the aforementioned points we also had great documentation since we created both Use Cases and User stories so there was no room for misinterpretation.

Sprint planning

Focused on delivering features and made choices on backend solutions. Tasks added to sprint:

- SearchView
- Results Swipe
- Instructions
- Topbar
- Routing

Sprint execution

During this sprint we were heavily focused on making the features that were described in the prototypes, using the user stories each team member took it upon themselves to pick up tasks and complete them within the sprint.

Features pull request to production

- SearchView
- Results Swipe
- Instructions
- Topbar

Reviewed pull requests for:

- SearchView
- Results Swipe

Instructions and Topbar have been postponed and will be added to the next sprint

Review & retrospect

Using a framework like React gave us the freedom to create features independently from each other with a good certainty that when merged it will function correctly. Also React has a strong reusable nature so team members could make use of components that were already in place. Now that some features were in place we needed to start setting up routing based on our diagrams. Also we chose to get food API access through our own backend not through React due to the need of higher flexibility.

#4 Sprint - 26st November 2nd December

This being our last sprint we saw that agile development also had its downfall. Some features didn't make it due to changes of scope or some were just too big to tackle in one sprint. This meant that this sprint was packed with improvements, fixes and tasks. Luckily we had the component structure of React to help us out. But that alone was not enough. The team had to schedule multiple meetings to pair-program the remaining features. On top of that user testing had been done on single features and needed to be done on the MVP in order to gather feedback to improve our application. The GitHub setup worked really well, but due to the stacking up of task we saw that this sprint was not well documented on the Scrum board. The tasks were not placed in the respected columns or even forgotten in the backlog. This was a good learning opportunity for us as a team, because we set out to work really structured and did that through most of the project, but in the last sprint with a bit more pressure it did not stand that strong. We can thank our preparation work for good eventual outcome.

Sprint planning

This sprint had the focus on refactoring and completing features from the previous sprint. Besides that we planned to finish the MVP and set that as the main priority. In order to have any form of user testing we needed the MVP to be done a bit more in advance. So we could show off the full MVP to our user test group and document their most relevant feedback.

Sprint execution

During the planning it was already clear that this sprint would be feature heavy, luckily due to React's reusable nature we could accomplish the tasks within the set time for the sprint.

Reviewed pull requests for:

- Profile
- BottomNav
- Results to Instructions
- Favorites
- History
- SearchView
- Instructions

Features merged to production:

- Profile
- Results to Instructions
- BottomNav
- Favorites
- History
- SearchView
- Instructions
- Bug fixes and CSS fixes

Review & retrospect

The amount of features delivered within this sprint was high, due to our approach of reusable components it was possible, but in a real business scenario we believe this would not happen. Also we decided that we should limit the amount of React dependencies, no need for component specific libraries, we should try to stick to the main one as much as possible. We chose react-md as our main component library.

Final learning moments:

Bringing all the experience from the sprints together we can say that we learned the following:

- Mac needs to make commits more frequently, code changes were too big per commit.
- Need to make sure the Scrum board is up to date.
- Modeling and prototyping was an super effective way to start and progress through a project. It gave us the handles that we needed to continuously develop.
- Features work well together, but many features released at once, could have been more gradual so we didn't have to review too much at once

3.1.12. Quality concepts

To make sure we build a high quality product, from all the different points of view/interest, we decided to use theoreticized concepts, that are carefully thought and will give us a good insight. We followed Roger Pressman's software engineering quality definition to describe the broad concept that we applied to our project: ***"An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it"***.

- the **effective software process** is represented by our implementation of Scrum for project management. It helped us keep things organized and work together, efficiently for the goal. You can find more about the process in Sprint Documentation chapter on page 38.
- we believe that we created an **useful product**, that can help our users with the task of finding cooking recipes. Regarding this, we had good feedback from a few users,

friends that we asked for opinions on the app, but there is of course room for improvement.

- the **value for those who produce it and those who use it** is represented by the experience and excitement for us to build the product, as for the users, the value stands in the functionality of our app and, on top, the pleasurable user experience provided

To cover more specific points, we used **Garvin's Quality Dimensions** as it follows:

- **Performance quality:** there is no visible lag or stutter on the app, and overall, it performs fast as expected. However, being a web application, its speed is highly dependent on the internet connection that we have no power on
- **Feature quality:** For the first version of the product, there will be a lack in this field. Only the core features will be included, but there are already planned ones to go in development.
- **Reliability:** As the our tests showed, reliability is on point. One big problem may be that we use data from an external API, which we don't control, and this may hold it down sometimes. We plan to develop our own recipes API in the future.
- **Conformance:** The app look and performance is on pair with current web standards. As everyday users of websites and apps, plus having the experience of developing some, we have a good view over today's level.
- **Serviceability:** We planned a lot ahead, and designed before starting to code, which made the source readable, easy to maintain, available for fixes and improvements at any time.
- **Aesthetics:** This point is highly subjective, but we believe that we made good visual design choices from the beginning, and while it can be improved of course, we are proud of the app look as it is now.
- **Perception:** It is dependant mostly on the users, but we did our best to make our app's mission easily perceived. The idea is quite simple and while we build more features, we will make sure to carefully blend them in smoothly, so users find their way with no difficulties.

Besides the previous two theoretical models we also used the widely know model from Hewlett-Packard called FURPS¹⁶. This model helped us define and separate our functional and nonfunctional requirements set by ourselves.

- Functionality
 - Capability - The current capability of the application is to generate a 'n' amount of results from a pool of 350.000 served by the Spoonacular API.
 - Reusability - The amount of searches that a user can do is limited to the amount of API calls we are allowed to make.
 - Portability - The application can be used on any device that has a user agent and an internet connection.
 - Security - We have basic authentication and a password policy set up, everything is currently saved in our backend
- Usability(UX)

¹⁶ <https://en.wikipedia.org/wiki/FURPS>

- Human Factors - The swipe mechanic is the strongest human centered factor of the application
- Aesthetics - Clean, clear and open design makes it easy to navigate the application
- Consistency - Every view in the application is based on the same template to strengthen cohesion and consistency
- Documentation - There is no usability documentation other than the user manual in this report
- Responsiveness - The application is created with the mobile first approach, that being said it is not yet optimal on larger screens than mobile
- Reliability
 - Availability - The application runs as long as our servers run and we have access to our data source.
 - Recoverability/Survivability - Other than resetting our servers and expending the API call range there are no further measure in place
 - Predictability (Stability) - Due to its fairly non complex nature the application runs within our predictions
 - Accuracy - The API can still show wrong query results, that is something that has to be handled in the future
- Performance
 - Speed - Since its just based on a single GET request per search the application is fast
 - Efficiency - We only call the API when all search term are set, meaning that we have made use of our own file to generate possible searches. This reduces the amount of API calls and keeps the application fast
 - Resource Consumption (power, ram, cache, etc.) - This depends on the user agent used
 - Scalability - Future features i.e. user generated content will require the application to scale, the current architecture supports almost any type of scaling imageable
- Supportability
 - Maintainability - The Github repository will be strictly maintained by the current development team, any updates or changes will be made to the production branch and delivered to all users.
 - Testability - Due to component based development with react, new features can be tested outside of the production environment. This is possible for unit as well as user testing.
 - Flexibility - Component based development gives us the power to work on new features while not affecting anything critical and making sure that it fits our standard since we can build using existing components
 - Localizability - Not localized to any region, it's in English and available to anyone

Taking into account the above points, we strongly believe that our product is of high quality, on all the different dimensions and perspectives.

3.2. Software development

A large part of a software production process is its development. After the initial idea, deciding how to best manage the project, we needed to decide what we are going to use to make it and how we are going to code it. Starting off with the first paragraph summing up the technologies that we have used.

3.2.1. Technologies

In this chapter we list all the technologies used that made this project possible.

Operating system

We have developed this project using both Windows and Mac OS x. Since the application is a web based solution, we didn't need to choose only one operating system.

Integrated Development Environment

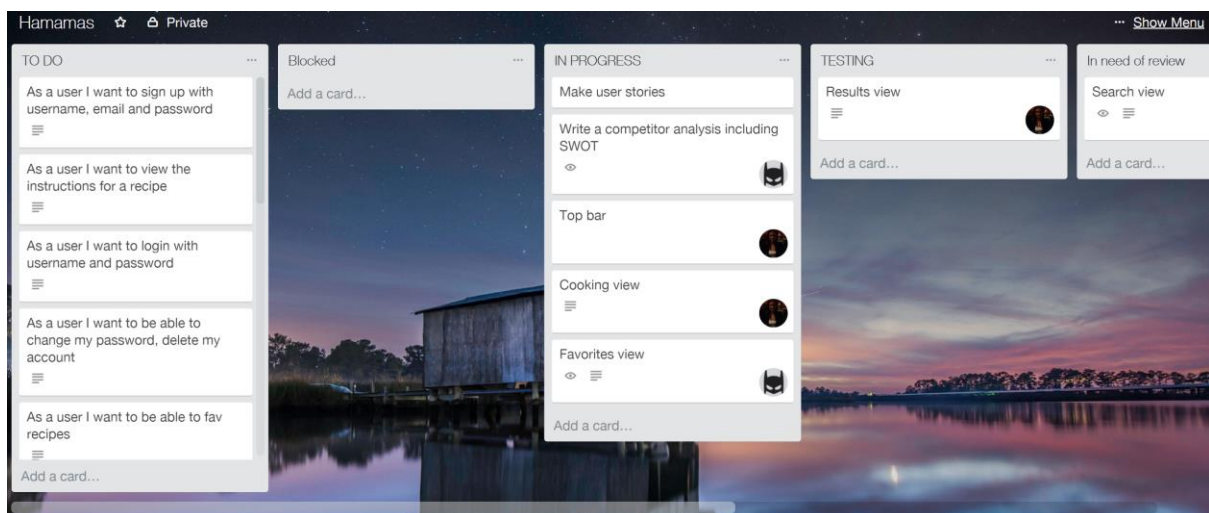
Visual Studio Code was used as the go to IDE. We choose this IDE because it had solid and simple integration to GIT, it had options to install external packages and it was available both on Windows and Mac OS x.

Programming language

We used Javascript with the React library. From the beginning we wanted to make a phone application. Mac was the only one that had experience with a native mobile language. So we choose to make it a web application that is designed for mobile. This way not all group members needed to learn an extra programming language and we covered both the iOS and Android market. Since both Dana and Petru were proficient with React and JavaScript was known by the whole team, we made the choice to use React as main framework for our application.

Project management

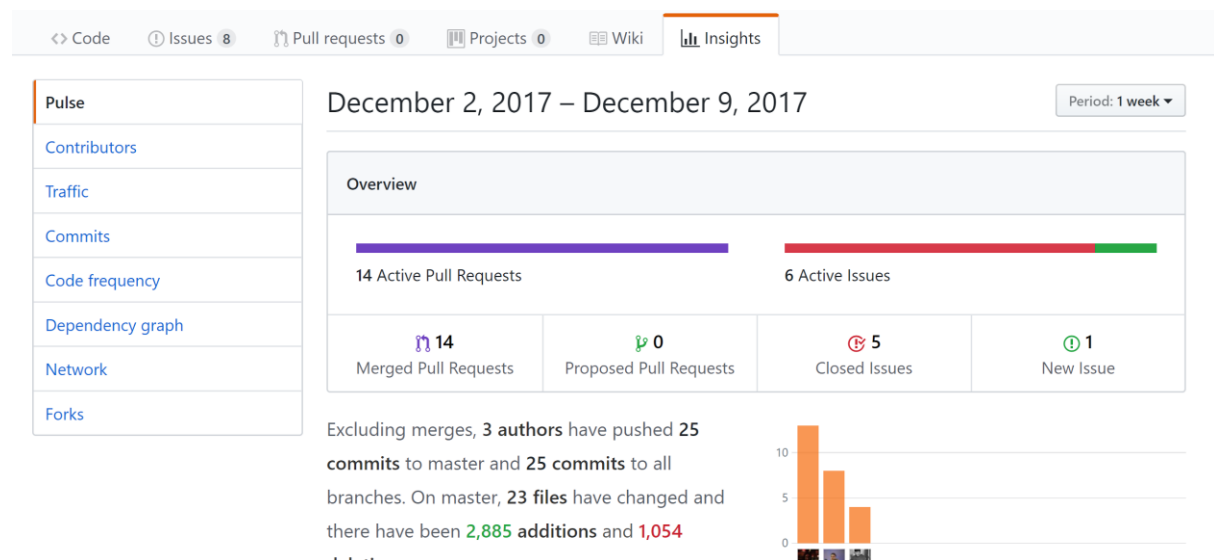
Trello has been used for project management. We needed a digital representation of a Scrum board. We choose Trello since the whole team has used it before and it is a great free alternative.



Screenshot of our Trello Scrum board

Version control

GitHub has been used for release management, testing and version control. The whole team has had previous experience with Git so we decided to use that solution. Petru was in charge of the repository and the whole team was in charge of reviewing pull requests and merging into the master branch. This process was not only effective, but super insightful since we reviewed the code together we found out that certain things like empty newlines would create unnecessary merge issues.



Screenshot of the Hamamas GitHub repository, "Insights" view

Recipe data

For the recipe data we make use of the Spoonacular API. We chose this one because it has a large variety of available recipes and it fitted best our use case. We are using two of their endpoints, one for generating recipes based on ingredients (we get back an array of recipes, which do not have instructions) and one for getting the cooking instructions for a recipe based on the recipe id. On top of this great functionality we signed up for an academic subscription which comes at the convenient price of \$5/month.

Design

Figma has been used for mockups and wireframes. Dana had previous experience with Figma, and her skills proved really useful for creating mockups and other design work. It was the main go to document while developing features and views, having a designer on the team gave us a huge advantage.

UML

For creating the use case diagram, system sequence diagram and sequence diagram we used Creately, an online UML editor which was easy to use and most importantly provides templates so we had a base to start from.

3.2.2. Progressive Web App (PWA) vs. Native App vs. React Native

For this project we decided to focus on mobile for two main reasons: firstly, it fits more this particular use case because users wouldn't want to carry their computer or tablet to the kitchen when cooking, it is much easier to check the recipe instructions on the phone; secondly, mobile web/application usage is leading by far compared to other devices, so we want to make sure we provide for the biggest audience first.

Having decided this we were then confronted with a difficult question: which approach should we take - progressive web app, native app or React native app? In the next paragraphs we will discuss what we took into consideration regarding the three options when we made a decision.

Native App

- creating a native app would have been difficult for us since only one of the group members has had experience with Java for Android development and none of the group members knew Objective-C or Swift
- even if we would have had the necessary knowledge we could have catered for only Android or iOS users, not for both, which is not what we wanted
- little or no code which can be reused in case we want to develop for desktop in the future, in that case we would have to use web technologies and nothing native

React Native

- this could have been a good alternative, but still requires some Java and/or Objective-C and Swift
- we could still not use the same code 100% for Android and iOS
- it would still not be reusable for desktop development, as in that case we would have to use React and not React Native

Native App + React Native

- a big problem with both of these is that they are found in Google Play or the App Store, thus the user needs to open one of these apps depending on their smartphone's operating system, search for the Hamamas app, select it, download it (this will of course occupy memory in their phone), wait for it to download, wait for it to install and only after that they are able to open the app and run it; this is a lot of work for a simple recipe search
- it has been proved that people use only a couple of apps daily, the rest are just installed in their phone but they don't use it, which we didn't want to happen for Hamamas

Progressive Web App

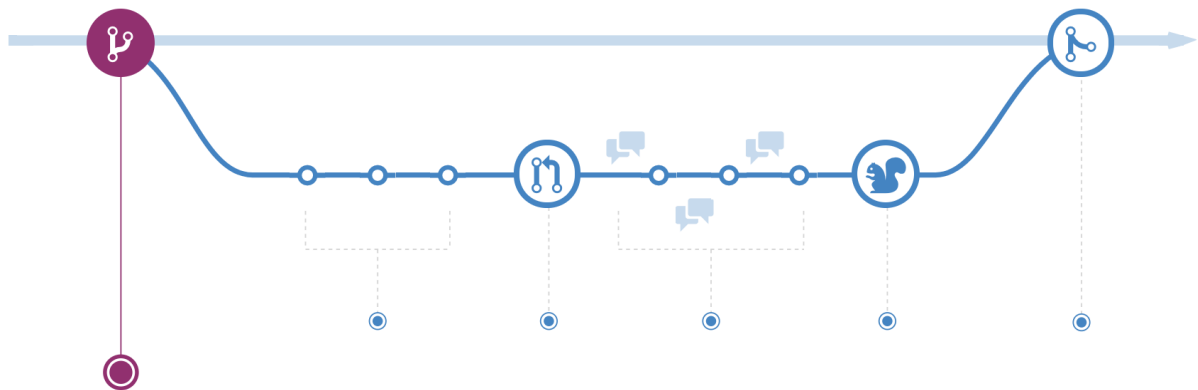
- no need to know native programming languages
- use web technologies like React, JavaScript
- it works on all platforms and devices, the only thing which needs to be tweaked is the CSS styling via media queries, no other changes needed
- no need to download from Google Play or App Store or install it
- deploy new versions seamless, without the user needing to update
- can share links to other people, for example if you want to send the cooking instructions of a recipe that the user really liked to a friend

Conclusion

Based on the above we decided to go for a PWA. To ease our work we used create-react-app npm package (similar to the Angular CLI), which is a boilerplate for creating React applications and does not require us to configure webpack and babel ourselves. It also comes with a service worker and a web app manifest¹⁷, so we basically do not have to do any additional work to transition our application to a PWA; any application generated with create-react-app is now by default a PWA.

3.2.3. Software Configuration Management

As mentioned, one of the benefits of building a Progressive Web App is the ability to deploy changes without the user needing to download an update. This is really important because you can fix bugs, or deliver new features blazing fast and to everyone (as some users don't update their apps). To take advantage of this and for many other benefits that come along, we needed a good strategy for the Software Configuration Management.

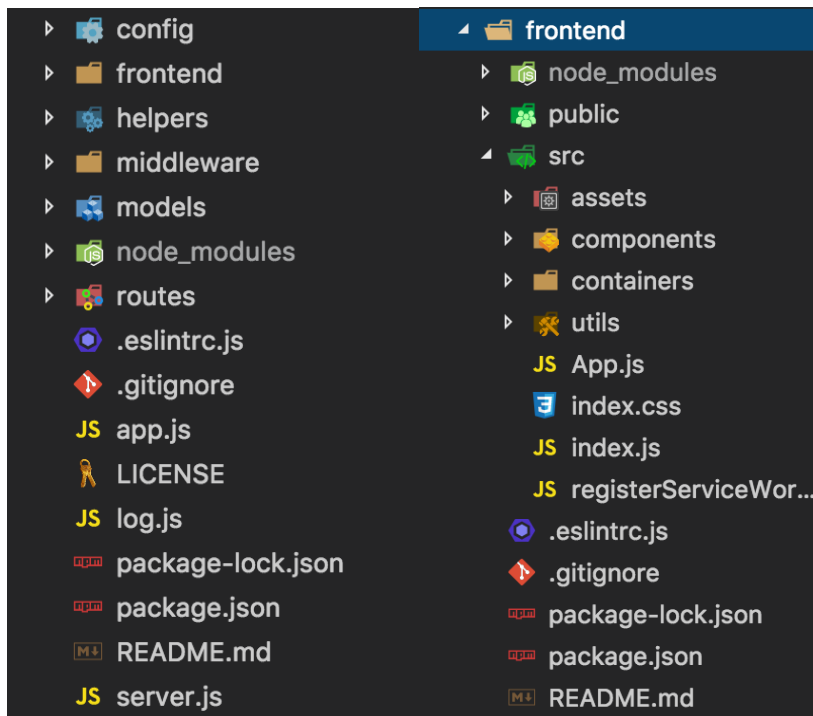


Github Flow overview, which we used and described below

We combined activities such as Identification, Change Control, Version Control, Configuration Auditing and Reporting into our development workflow. We used Git, Github and its amazing Github Flow¹⁸ to integrate these activities naturally into our process and make it easy to handle changes and versioning. The **identification** activity was conducted at all times by means of a well defined directory structure, using unique suggestive names for our files on a high level, functions and variables on the low.

¹⁷ <https://developers.google.com/web/fundamentals/web-app-manifest/>

¹⁸ <https://guides.github.com/introduction/flow/>



Project directory tree

As for the development process, it went as following:

Workflow

We have a remote git repository, hosted on Github¹⁹, where the whole team has access to the source code and its history. We work with branches, a core concept of Git²⁰, **master** being our main branch, always deployable. When a developer is working on a new feature or bug fix or anything that implies changes/additions to the code, he has to follow the steps (the whole process stands as the **Change Control** activity):

1. **Create a branch** - create a new branch, usually cloning **master**, that will be properly named according to what you are working on (e.g. **feature/login**, **fix/main-search-btn**). You will then work on this branch, this way not interfering with other's work

```
→ hamamas git:(master) git checkout -b feature/my-feature
Switched to a new branch 'feature/my-feature'
→ hamamas git:(feature/my-feature) █
```

Command for creating a new branch

2. **Add commits** - make changes to achieve the purpose of your task, don't forget to commit often, this way being able to go back if needed. Add descriptive messages to your commits, so everyone (even you after a while) can quickly understand what happened, what does your change do (a good way of **Reporting**).

¹⁹ <https://github.com/>

²⁰ <https://git-scm.com/>

3. **Open a pull request** - are you done? have you tested your changes multiple times? then it's time to ask for review. Push the local changes of your branch to the remote repo, go on GitHub and make a pull request. The repository is set so you will need at least one acceptance review to be able to merge the changes into the **master** branch.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base: master ← compare: feature/my-feature ✓ Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Dec 17, 2017

petr166 added change comment #BE | app.js 7b7253c

Showing 1 changed file with 2 additions and 0 deletions. Unified Split


2 app.js View

```
@@ -31,3 +31,5 @@ app.use('/recipes', recipesRoutes);
31 31 app.use(errorMiddleware);
32 32
33 33 module.exports = app;
34 +
35 +console.log('I made a change!!');
```


GitHub's compare changes prior to creating a pull request

4. **Discussions** - your colleagues have to go through your changes, make suggestions on what can be improved, ask questions about your code, anything that leads to a better/more understandable version of what you have done (we achieve **Configuration Auditing**). During this step you can and are encouraged to add commits to your branch (don't forget to also push them to remote) to resolve discussions and get the approval from your colleagues
5. **Integrate** - your pull request got approved by your mates and now it should be integrated with the source code into the **master** branch (the history of the branches will be merged, expressing the **Version Control**). Two things can happen:
 - a. Your branch has **NO** conflicts with **master**, so it can be automatically merged. In this case all you need to do is to press "Merge", add a commit message and voilà, your changes are now integrated with the source code. Take a new task and repeat the process.
 - b. Your branch has conflicts with **master**, meaning that it cannot be automatically merged. In this case you should go to step 6



added change comment #BE | app.js #47



 **Open** petr166 wants to merge 1 commit into `master` from `feature/my-feature`



Conversation 0 Commits 1 Files changed 1

 petr166 commented 25 minutes ago + 🗨️ ✎️



No description provided.


  added change comment #BE | app.js 7b7253c

  petr166 requested a review from Machess 25 minutes ago

  Danalliescu approved these changes 2 minutes ago View changes

Add more commits by pushing to the `feature/my-feature` branch on petr166/hamamas.

  **Changes approved** Show all reviewers
1 approved review by reviewers with write access. [Learn more.](#)

 **This branch has no conflicts with the base branch**
Merging can be performed automatically.

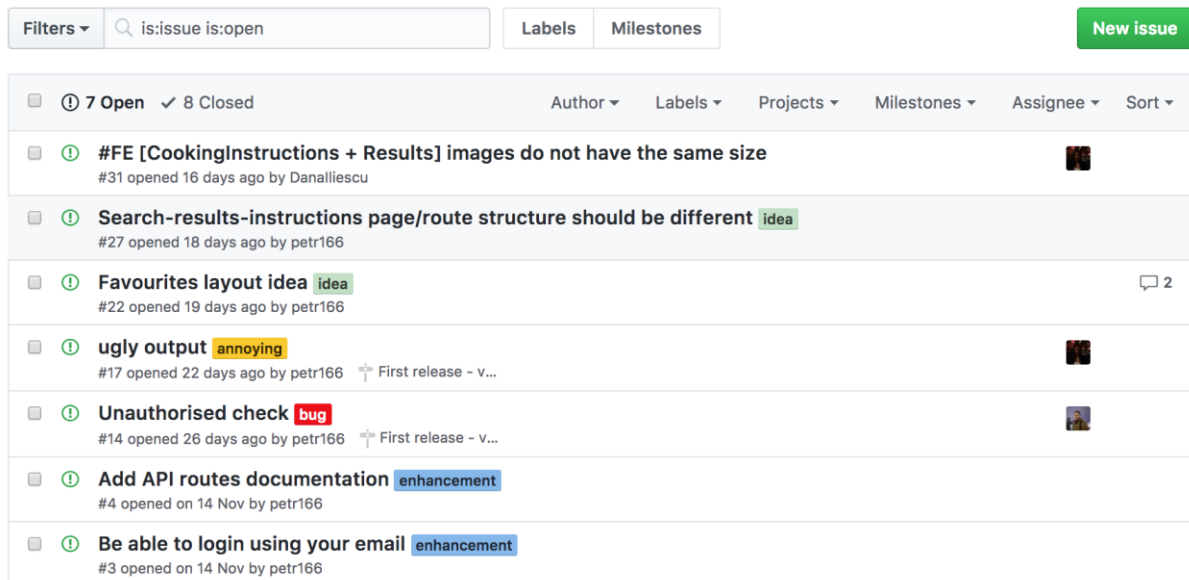
Merge pull request You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Example of a pull request with approved changes and no conflicts

6. **Fix conflicts** - if the parts changed by you were altered during on **master** by the time you finished your task and made a PR, Git may get confused and not know which changes to keep, so it asks you to decide:
 - a. On your machine, get the latest changes from the remote (git pull), especially the changes from **master**
 - b. Make sure you have your task branch checked out (git checkout your-branch), now merge **master** your this one (git merge master). This will include the conflicts.
 - c. Go to the code, in the files that have conflicts and pick the changes to be kept, removing the other version (sometimes you might have to keep them both or make a slight change to accommodate your task goal!).
 - d. After you fixed all the conflicts and made sure to test the app, with the functionality added by you, commit and push the new changes
 - e. Now, when you go to the PR page you should see the green “Merge” button, do as in step 5

We also use the repo’s “Issues” page to flag and discuss bugs that we find or to add ideas that might be implemented. Using as much as Github has to offer has the benefit of having

everything in one place, plus nice integration between the features. It makes our software development process easier and more enjoyable.



Screenshot of the Issues page on Hamamas repository, 7 open issues and 8 closed

3.2.4. User testing

We have decided to let users outside of the development team test the application in order to see if there are any UX issues or features that might have been overlooked. The aspects we looked at when testing were learnability, efficiency, memorability, errors and satisfaction.

Learnability

We asked the testers to fulfil different tasks using our application, as for example register, login, search, change their password or delete their account, whilst we stood by and took notes. Even if it was the first time they were using the software, our subjects didn't show any difficulties using the application, navigation seemed easy, and the feedback from the system assured them that they were on the right track.

Efficiency

We could notice that compared to the first time use, after they got accustomed with the application they were more efficient and faster in completing the tasks.

Memorability

Due to a simple and intuitive user interface, users were able to re-establish proficiency fast, even after some time has passed (we conducted the same test a few days after).

Errors

Subjects ran into problems most when they had to create a new account as they did not know they had to have a password which has a minimum length of six characters and when they tried to hit the "Feed me" button even though there were no chosen ingredients. In this case a snackbar message popped up suggesting that they first have to input some ingredients, so we can say that they recovered from this error easily.

Satisfaction

These users stated that they are overall content with the application, that it is easy to use and that they consider the swiping as a cool feature. We should of course run this test on a larger amount of people so that we can get more feedback thus more accuracy on the usability of the product.

Feedback

We have received the following feedback from our test users:

After the first sign in, currently the app just displays the search. It would be a great improvement to create a small paragraph (alert) describing the usage of the application to first time users.

Signup does not prompt for password policy. When signing up, the user does not receive a message saying that the password needs to be at least 6 characters long.

There is no way of doing a random search. It would be a nice addition to have a random ingredient search option. In practice this would mean either hitting the search button without a search or adding a random search option that just generates a few search terms.

3.2.5. Code snippets

We will present a few parts of the app the we think are interesting because of the problem that they solve, or the approach taken on them.

The DB connection handler

```
JS mongo.js x
Petru Birzu, a month ago | 1 author (Petru Birzu)
1  const mongoose = require('mongoose');
2  const Promise = require('bluebird');
3  const log = require('../log');
4
5  mongoose.Promise = Promise; // plug-in bluebird as mongoose Promise
6
7  // connect to mongo host, set retry on initial fail
8  const connectMongo = (config) => {
9    mongoose.connect(config.host, config.options)
10     .catch(() => {
11       setTimeout(() => { connectMongo(config); }, 2000);
12     });
13  };
14
15  // to export: init mongo connection, set logging
16  const start = (config) => {
17    return new Promise((resolve) => {
18      mongoose.connection.on('open', () => {
19        log.log('mongo', `connected to db: "${config.host}"`);
20        return resolve();
21      });
22      mongoose.connection.on('error', (err) => {
23        log.err('mongo', 'error', err.message);
24      });
25      connectMongo(config);
26    });
27  };
28
29  module.exports = { start };
```

This file (“/config/mongo.js”) takes care of the MongoDB connection that happens when the backend starts. The interesting part is, on the “start” function, it returns a Promise, meaning that it will run asynchronously and can be chained to other Promises to sequentially execute pieces of code according to our needs. How it works is that it will try to connect to MongoDB and retry on failure every 2 seconds. In “server.js” we call the “start” function and chain the Server start after a successful database connection. This is very important as there will be a lot of errors if we start handling HTTP requests without having a database connection. As db connection errors don’t happen often in development because the db is usually hosted locally, this may not seem like a problem, but in a production environment the db may, for some reason, not be available, thus making your app fail on every request.

Handle user registration

```
38 UserSchema.statics.createUser = (data) => {
39   if (data.password.length < 6) {
40     return Promise.reject(new Error('The password should have at least 6 characters'));
41   }
42
43   if (!emailValidator.validate(data.email)) {
44     return Promise.reject(new Error('The email is invalid'));
45   }
46
47   const user = new User(data);
48   // hash the user's password before saving
49   return crypto.getHash(user.password)
50     .then((hash) => {
51       user.password = hash;
52       return user.save()
53         .then((savedUser) => {
54           return {
55             ...savedUser._doc,
56             password: undefined,
57           };
58         });
59     });
60   };
```

In the user model (“/models/user.js”), this is the static method used to handle user registration, adding a new user to the database. It has validation checks for password and email for which we use “email-validator” npm package. Now the interesting part is with the password that gets saved in the database. There is one step before saving the new user, and that is encrypting his password. Why so? Well, in a perfect world this would not be necessary, but if the db gets compromised, and it could, it will be much harder to get access to the user’s account because the actual password is not there.

```
8 // get a hash value, use to hash passwords before saving to db
9 const getHash = (value, saltRounds = 10) => {
10   return new Promise((resolve, reject) => {
11     bcryptjs.genSalt(saltRounds, (err, salt) => {
12       if (err) return reject(new Error('Salt generate failed'));
13
14       bcryptjs.hash(value, salt, (err1, hash) => {
15         if (err1) return reject(new Error('Hash generate failed'));
16         return resolve(hash);
17       });
18     });
19   });
20   };
```

This is the function used to encrypt the password. It uses “bcryptjs” npm package.

Recently viewed list

```
34  /**
35   * add an item to recently viewed recipe list
36   * @param {object} itemShallow - the item to be added
37   * e.g. { id: '1234', name: 'Nail soup' }
38   * @return {boolean} - success
39   */
40  const addToRecentList = (itemShallow) => {
41    const item = sanitizeItem(itemShallow);
42    let recentList = getRecentList(true);
43    const existingRecipe = searchRecipe(item.id, recentList);
44
45    if (existingRecipe !== -1) {
46      // remove if existing
47      recentList.splice(existingRecipe, 1);
48    } else if (recentList.length >= RECENT_SIZE_LIMIT) {
49      // remove the old one if size limit reached
50      recentList = recentList.slice(1);
51    }
52
53    recentList.push(item); // add the new item
54
55    try {
56      localStorage.setItem(RECENT_KEY, JSON.stringify(recentList));
57      return true;
58    } catch (e) {
59      return false;
60    }
61  };
```

For the frontend React app, in “/frontend/src/utils/recent.js” this is the function used to add a new item to the recent list. We decided to take advantage of the browser’s built in `localStorage` to implement the “recently viewed” functionality. This is how it works: it takes an object and stringifies its properties with “`sanitizeItem`”, so you don’t have errors saving it to the `localStorage` later. Then you pull the current recent list, search for the new item, to make sure it doesn’t exist (if it does we remove it). Another handled case is if the size limit is reached (we decided to keep the recent list to a limit of 50 at first, then see if this value works for our users). After that the item is added to the list, the list gets stringified and stored in our local storage. In the recent page view it gets pulled and displayed.

App state dependent view

```
JS Profile.js x
167 render() {
168   const { isFetching, error, accountDeleted } = this.state;
169
170   let Content; // conditionally show in return
171   if (isFetching) {
172     Content = (<CircularProgress id='profileLoading' />);
173   } else if (error) {
174     Content = (
175       <div style={{ textAlign: 'center' }}>
176         <b>Error:</b> {error}<br />
177         <a href='#!' onClick={this.fetchUserData}>Try again</a>
178       </div>
179     );
180   } else if (accountDeleted) {
181     Content = (
182       <p>Your account has been deleted. You will be redirected soon...</p>
183     );
184   } else {
185     const {
186       user: { name, email, username }, dialogVisible, dialogField, toasts, modalVisible
187     } = this.state;
188     Content = (
189       <div>
190         <h1 style={{ textTransform: 'capitalize' }}>{name || username}</h1>
191
192         <InfoLine>@{username}</InfoLine>
193         <InfoLine>{email}</InfoLine>
194       </div>
195     );
196   }
197   return Content;
198 }
```

This example is from “/frontend/src/containers/Profile.js”, in the “render” method. It shows the convenience of using React’s state management to conditionally display the content that the user needs to be aware of. In the Profile screen, first of all “isFetching” state key is true while we are getting the user data from the server, thus displaying a loading icon. In case an error occurs, we display the error message and give the user the possibility to try to fetch the data again (maybe the connection was down). When the user chooses to delete his account, the “accountDeleted” key turns true, thus displaying a confirmation message and redirecting the user in a few seconds. And finally, if none of the abnormal cases are true, the “Content” displayed contains the user information. This is an elegant way to handle the different states your app/part of it may be in, and it is the very purpose of React. We use similar approaches on all parts of our app, and it works like charm.

3.2.6. Maintenance



Maintenance begins as soon as a software product is launched into production, which for us didn’t happen yet. However in development we did our best to minimize the amount of work we will need to do after we deliver the product.

We tested our application as much as possible trying to find bugs or unexpected behavior. The most problems came actually from the Spoonacular API, namely inconsistency of data. We would get errors for some recipes that did not have instructions or ingredients which had the same id. It is of course impossible to try and test all the recipes that they have in hope of

finding this type of inconsistency, but for example after we found out about the problem with the same id for ingredients we added an unique id for the instructions as well in order to prevent errors (even if we didn't encounter any).

When developing we worked each on our branch and then merged into master when we were sure the code was tested and worked bug-free, but sometimes it happened that we discovered a bug only after it was integrated in the master branch (as mentioned above we didn't know which recipes are in the right format and which not). We created pull requests for every bug we discovered and fixed it, as you can see in the examples below.

 **Added uuid for both ingredients and instructions** 
#46 by Danalliescu was merged 10 days ago • Approved

 **fixed error when there are no instructions** 
#44 by Danalliescu was merged 10 days ago • Approved

Screenshot from the GitHub repository on pull requests for fixing bugs.

Another way to minimize maintenance work was to follow code standards and good code practices, which will in the future will help us save time by easily understanding our own and others' code. As the programming expert Martin Fowler said: "Any fool can write code that a computer can understand, good programmers write code that humans can understand". We tried to have that in the back of our heads at all time.

Using React means that we have a component-based architecture in the frontend part of the application, which means we have modularity. This will be an advantage when we will want to change something in the code, because modifying a component will not alter other parts of the system. We tried to implement the same concept in the backend by having several JS files according to what the code inside them is related to.

In the future we expect to deal with corrective, adaptive, perfective and preventive change²¹.

Corrective Change

As the app is going to be more and more used, more bugs or unwanted behavior will probably be discovered by our users, which we will have to fix.

Adaptive Change

We will have to keep an eye on our software dependencies (React, Node, Babel, Webpack, react-md, styled-components to mention just a few) as these can have breaking changes when releasing a new version and API endpoints which can also be modified. This way we can adapt our code so that it fits the new environments and works without throwing errors or warnings.

Perfective Change

Probably we will put the most work in perfective change, adding new features or improvements, or on the contrary remove features that prove to be not efficient. We will learn all this as the application is more exposed to users.

²¹ <https://endertech.com/blog/maintenance-bug-fixing-4-types-maintenance>

Preventive Change

We will try to optimize our code even more and make sure the documentation is up-to-date, which in the long run means better readability and understanding of code and easier maintainability.

4. User Manual

Even though our app is pretty easy and intuitive to use, we made a basic user manual which explains how to perform different tasks.

Create a user account and login

1. open the website and input your a valid email address, a username and a password
 - a. password must be at least 6 characters long
 - b. if the username is already in use, you will be asked to choose another one
2. click the sign up button, you will now be redirected to the login page where you can input username and password
3. click the login button, you will be redirected to the search view

Search recipe based on ingredients

1. open the website and login
2. choose which ingredients you want to use for the recipe in the search bar
 - a. chosen ingredients will appear inside the search bar, if wanted you can remove them by clicking the “x” icon
3. click the “Feed me” button
 - a. if no ingredients are chosen you will be asked to choose at least one



Directions for search view

Browse results

1. search recipe by ingredients, you will be then redirected to the results view
2.
 - if you swipe *left*, another result will be displayed
 - if you swipe *right*, you will be redirected to the cooking instructions for that recipe
 - alternatively you can use the "x" button (left) or "check" button (right)



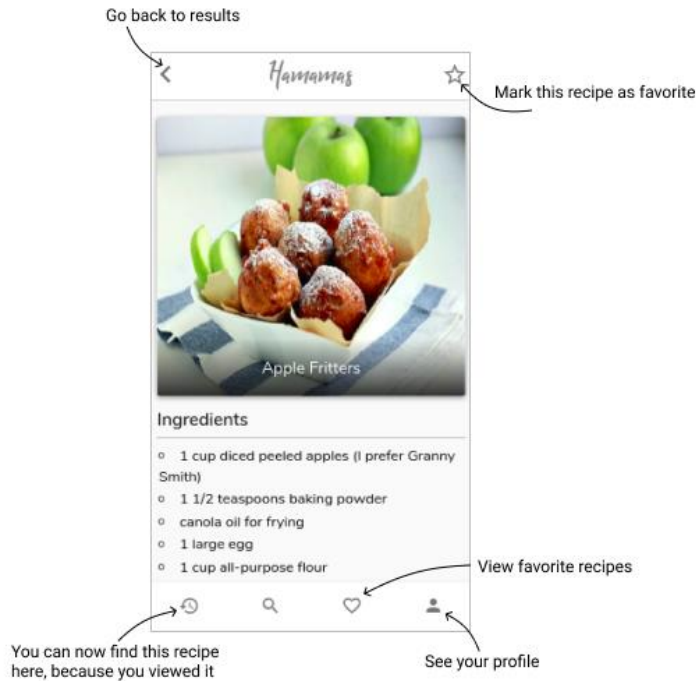
Directions for results view

See cooking instructions for a recipe

1. search recipe by ingredients and swipe right/choose one of the result recipes
2. view cooking instructions
3. optional: you can now mark a recipe as favorite/unfavor it or, if you decide to make it, click on the "I made it!" button

See a favorite recipe or a recently viewed one

1. login
2. press the heart icon (favorite) or the history icon (recently viewed) in the bottom navigation bar, you will be redirected to the specific view
3. click/tap on one of the recipe titles, you will be redirected to the cooking instructions
 - a. if you do not have any favorites or you haven't seen any recipes yet, click the "start" button that is displayed, you will then be redirected to the search view where you can search, view a recipe and afterwards mark recipes as favorite



Directions for cooking instructions view

Change password

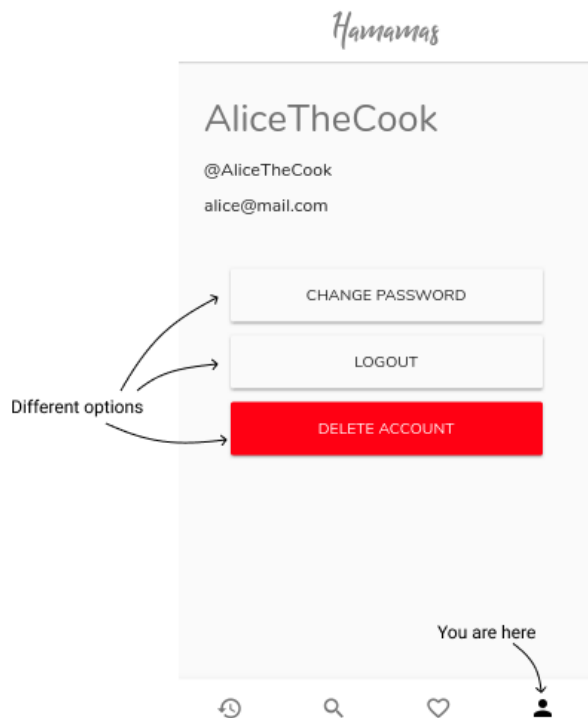
1. login
2. press the user icon in the bottom navigation bar
3. click the change password button
 - a. you will be prompted to input a new password
 - b. click ok, your password is now changed

Delete account

1. login
2. press the user icon in the bottom navigation bar
3. click delete account
 - a. you will be prompted a message asking if you are sure you wanna delete account
 - b. if you click "yes", your account will be deleted and you will be redirected to the landing page
 - c. if you click "no", the popup message will close and you will still be on the profile view

Logout

1. login
2. press the user icon in the bottom navigation bar
3. click logout button, you will be then redirected to the login view



Directions for profile view

5. Usability Heuristics Evaluation

After the first release was done we assessed our product against Jakob Nielsen's usability heuristics²² as we wanted to determine the application's usability from a qualitative point of view and see if it actually fulfils the user's needs. We looked at each of the ten principles and how they apply to the Hamamas app.

1. Visibility of system status

The user knows at all times what is happening when they use the application through appropriate feedback. This feedback includes for example the loader after you click "Feed me" expressing that clicking the button really did something and the user should wait for a second until fetching the results. Another example would be that when the user marks a recipe as favorite the star icon becomes filled (it is not only the border anymore), or when the user marks a recipe as made, the button becomes disabled (since you cannot unmake a recipe). This principle applies also in the case of an user deleting their account, there is a screen showing that their account has been successfully deleted. It is important to show status and give feedback as users get a the feeling that they are on the right track and their actions actually mean something.



Example of showing status - before clicking/tapping the star and after

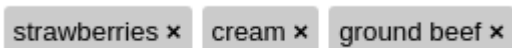
²² <https://www.nngroup.com/articles/ten-usability-heuristics/>

2. Match between the system and the real world

The language we are using is accessible and easy to understand for everyone, even for people who are not used to cooking. Besides food related terms (“recipe”, “instructions”, “ingredients”), we use terms that are common to every application such as “account” and “profile”. The icons used in Hamamas are also known metaphors such as a magnifying glass for searching, a heart for favorites, a person icon for profile, and a back arrow for navigating to the previous page.

3. User control and freedom

We want users to be able to undo actions if those actions were made by accident. For instance if they mistakenly select an ingredient, it wouldn’t make sense to be stuck with that ingredient for the search. That is why we added buttons on each added ingredient so that they can be removed if not desired. Imagine if the user clicks/taps by mistake the “delete profile” button, they will delete their account even though that was not their intention. That is the reason why we added a confirmation modal so users can actually cancel this action.



Example of user control and freedom - user can delete an ingredient added by mistake

4. Consistency and standards

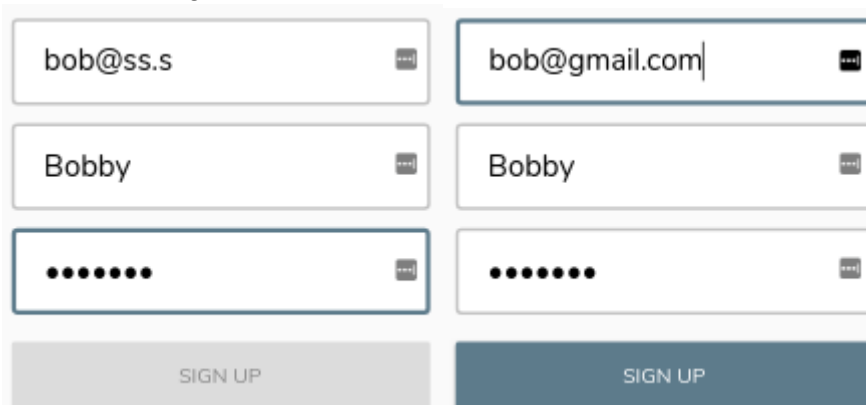
Our application is consistent throughout different views, for example we have the logo on both signup and login views (they both have kind of the same layout), and all views have the bottom navigation in the same place and navigation icons are in the same order. We follow standards such as having the “back” button in the top left corner of the screen also the profile section is usually on the right side of the screen.



Example of following standards - back button is on the top bar on the left side

5. Error prevention

A good UI prevents user making errors. In our case, on the register screen the sign up button is disabled until the form is valid, so users cannot click it. This means the email address is a valid one, username is at least four characters long and password is at least six characters long.



Example of error prevention - if the email is invalid then the sign up button is disabled, but with a valid email address it can be clickable

6. Recognition rather than recall

This principle implies that the user should not need to remember information from one view to another. To avoid this we have a clear distinction in the bottom navigation between the view the user is on (the current route icon is in a dark color) and the other views (which are of light grey color), so the user can see at a first glance where they are at.



Example of recognition over recall in the bottom navigation

7. Flexibility and efficiency of use accelerators

This application is a recipe generator so its main point is to generate recipes based on the ingredients that the user is inputting. However we have a few features for a more advanced user such as marking a recipe as favorite. This principle will be enforced more when we will implement filtering and sorting features as these are a good example of efficiency accelerator.

8. Aesthetic and minimalist design

Luckily we all have a tendency for minimalist design so this principle was pretty easy to follow. We created our UI based on simplicity and elegance. We believe that stripping down the views of content that is not a must will only benefit the user as they can concentrate on their task without any distractions.

9. Help users recognize, diagnose and recover from errors

Displaying appropriate messages when errors occur is a critical aspect in any application. Choosing terms the user can understand (not codes for example) and suggesting what went wrong can help the user easily recover from errors. In this case we are using either small paragraphs displaying the error (for example if username and password don't match) or something called snackbar, which is a small message that pops up describing the error and which disappears after a few seconds (we are using snackbars for a search without any ingredients, so we are telling the user he/she needs to input some ingredients before any results can be generated).

Even though it is not as severe mistake, we are displaying the errors from the food API as they come, so if the error from the API contains codes or technical terms we will display it. We have a "try again" button which will make a new request to the API so recovery is reassured.



Example of recovering from errors - our error message vs. Spoonacular API error message

10. Help and Documentation

We are providing a user manual which can help the user when in doubt about how to carry out any of the features of this application. The user manual can be viewed at page number 60 in this report.

Conclusion

Based on the above usability heuristics we can conclude that our application is indeed easy to use and learn, as well as enjoyable.

6. Possible improvements and new features

Improvements

Good improvements to the present application will be to make it responsive, so it looks good on tablet and desktop as well, and displaying the time it takes to cook the recipe together with the title and picture. This will be a nice touch so users don't have to swipe right on a recipe only to read the instructions and find out that it will take a lot more time to prepare it than they are willing to spend.

In the future we would like to make a small change to the layout of the application by adding a view where there are three tabs at the top: made recipes, favorites, and recently viewed. The user can then switch between these by clicking on one of them. This way we can have components that are very similar in the same place.

Possible new features

Regarding the new features we believe that so many cool things can be done in the future with our application and we are looking forward to begin starting on it again soon.

Filters

For example, we would like to add another view called "filters", where the users can toggle on and off filters such as dairy free, gluten free, vegan and vegetarian. There can also be checkboxes for displaying only recipes that take under 15 minutes, 15 - 30 minutes, 30 min - 1 hour or more than 1 hour. Another filter that can come in handy is the dish type, which can be starter, main course or dessert. Adding filters to the app will significantly improve the user experience.

Different sorting options

Sorting results based on the time to cook is another feature we would like to implement, so the user can choose to have displayed first the recipes which take the least time to cook.

Also adding a difficulty rating would improve the results for users, maybe they would like a challenge or don't have too much time and want to create an easy meal.

Saving searches

Another idea would be to give the user the possibility to save a query (meaning chosen ingredients + filters + sorting options) and make it default if wanted. This would save time for people who for example are vegan (so they will always have the vegan filter checked), want

to cook something as fast as possible (sort based on least time to cook) and have always certain ingredients that they use.

Social Aspects

We think that the social aspect is missing in the available cooking applications and websites, so we definitely want to add a chat function. People who have made the same recipes (meaning have clicked the “I made it!” button on the same recipes) will then “match” and be able to chat and share impressions and thoughts about the recipe, how delicious their dishes turned out to be (or not), etc. To enforce the social aspect even more we could add a feature for enabling location, then the user will be displayed only people with whom they matched from their vicinity, so eventually they could meet up and cook together.

It will be also interesting to see what your “matches” or “friends” have cooked and their favorite recipes. Grouping users would also be beneficial in the future, imagine a student dorm with communal kitchens. When you set up a group in Hamamas you can add any user you would like to receive a notification based on what you have cooked so you can instantly invite them to come over or at least let them know that there's enough food that it can be shared around.

User generated content

Currently we rely on an API to get recipes, this is something that might change in the future especially if we use the power of user generated content and let users add and edit their own recipes. this will of course require a form of input and more data handling on our end, but that would eventually happen when the application will need to scale up.

Random recipe or most searched recipe

Even though this is meant to be a recipe generator based on ingredients you already have at home, in the future we could make it so that the user gets some recipes even though they do not choose any ingredients, which could be random recipes, recipes that are similar to the user's favorites or recipes that the user's “matches” have made or marked as favorite.

Save on food expenditure

One of our initial ideas was to reduce food waste so users use their current pantry for searches. In the future we would like to give the users the ability to see where the ingredients they want to use are the cheapest or organic. Making the application not only for people that want to reduce food waste, but also for those that would like to save on food expenditure.

Not only useful for consumers

Continuing on the previous feature idea, we also so implications for restaurants. Since not only consumers are wasting food on a large scale. We believe that our application could one day prove useful to restaurant owners by showing the dishes that are still available that day. This means that customers could either chose their favorite ingredients that the restaurant has and send their request, but also the other way around that the restaurant only displays what they can make on that particular day so they will actually finish all their fresh produce and other short shell life products.

Video cooking tutorials

Something that we believe would become even more popular in the future would be actual videos that exactly shows how a dish is done and can be manipulated (paused, slow motion, using subtitles). This would mean that users would be able to search based on any ingredient they like, receive recipes based on their search and then have an option to either see video or just picture instructions. This idea is currently too idealistic since there is no good way of receiving such content and especially linking it to ingredient searches.

OAuth via other social media

To help users save time we could also implement authentication via social media, for example Instagram, Facebook or Twitter.

7. Reflections

7.1. How Scrum + XP worked

Scrum and XP have proved to be a successful combination for our team. We managed to follow the parts we decided to use from each (Sprints and all the meetings, user stories, backlog prioritization from Scrum, quick design, pair programming and tests from XP). We saw the value in each one of them, even if at the beginning it seemed like a lot to deal with.

We found especially interesting and enjoyable the pair programming sessions and the Scrum meetings. By developing together we were more efficient and we made fewer mistakes, thus we saved time and we produced quality code. The Scrum meetings were useful as well, because during the retrospective and review we had a good opportunity to see how the sprint went and how we can improve it, so the process went from good to better each time.

By using Scrum and sprints we could have a clear idea about where we are at in the project, what has been done and is still left to do, so we could make sure we are on schedule. Prioritizing our tasks ensured that we deal with what is more important first and that we don't miss out on anything.

7.2. Professional & Personal development

Looking back we realize now how much we have learnt from this project both professionally and personally.

We got to create a project from scratch and engage in every single stage of it, from requirements gathering, prototyping, designing, testing, developing to the final pull request we made. We applied different software design practices such as Risk Management, Maintenance and Quality Concepts. We were an agile team and we deepened our knowledge about particular libraries such as React and programming languages such as Node.js, which nowadays are extremely popular. We learnt how to choose between and use different external libraries and APIs for the frontend part, since we used packages for swiping (react-swipe-card), auto-complete search (react-chips) and UI components (react-md). More than that, we got involved in the community by submitting issues to these

packages' GitHub repositories asking for information or when discovering unusual behavior. We are sure that all of this will be useful in our future careers in the world of software development.

Improvements can be seen in the soft skills department as well. We valued good communication, helping each other, patience, seriosity and professionalism, but also a positive and friendly attitude which made it easier to cope with the project's struggles.

8. Conclusions

In this final chapter we combine our learnings and experiences from this project into two conclusions. We chose to do this because both the product and its process were separately important enough to take into account for future endeavours.

8.1. Product Conclusion

The application came together in ways we did not deem possible from the beginning, that being said we still would have liked to add more features, especially the ones related to connecting users within the application. We have managed to build a software product which helps users search recipes by ingredients (recipe generator) that they can browse by swiping; they can view a recipe's cooking instructions and mark it as favorite if wanted. There is a view for favorite recipes and one for recently viewed recipes (history). As any other application has feature like register, login, logout, change password or delete account. To conclude, overall we are proud of the current version of Hamamas and will try to implement more features in the future.

8.2. Process Conclusion

With our team being so diverse in knowledge and programming skills it was interesting to get each other to a level that everyone could participate equally. The team did a great job of including everyone and meeting up as much as possible in order to always be aware of any issues or to explain changes to the codebase. This showed that our planning was flexible enough to integrate sessions that were not always product related but beneficial for the team. In conclusion the success of the Hamamas team was definitely the clear and frequent communication between the members and the clear definition of engineering practises that were used to develop at a fast pace.

Special thanks

We would like to mention and thank other people indirectly involved in the project, thank you for helping in with our project!

User testers:

Johannes Elton Hansen
Gudny Valborg Gudmundsdottir
Behnam Farrokhi
Morten Ulrich

Lecturer:

Christian Ole Kirschberg

API data:

Spoonacular API

Bibliography

- All previous reports made by either Dana, Petru or Mac
- Reel, J., "Critical Success Factors in Software Projects", *IEEE Software*, May 1999, pp. 18-23
- Roger S. Pressman & Bruce R. Maxim, *Software Engineering A Practitioner's Approach*
- <https://stackoverflow.com/>
- <https://www.upwork.com/hiring/development/agile-vs-waterfall/>
- <https://www.figma.com/>
- <https://react-md.mlaursen.com/>
- <https://reactjs.org/>
- <https://www.npmjs.com/>
- <https://eslint.org/>
- <https://github.com/>
- <https://www.w3schools.com/js/>
- <https://www.javascript.com/>
- <https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb>
- <https://code.visualstudio.com/>
- <http://brandoncoppernoll.com/project-management/>
- <http://www.supercook.com/#/recipes>
- <https://www.forbes.com/sites/work-in-progress/2011/07/12/be-your-own-brand-champion-or-get-one-now/#3e0f5d564b43>
- <https://www.madebymany.com/stories/operational-feasibility>
- <https://www.upwork.com/hiring/development/agile-vs-waterfall/>
- <https://www.uxpin.com/studio/blog/bad-ux-makes-users-blame>
- <https://iq.intel.com/how-swipe-left-swipe-right-became-a-cultural-phenomenon/>
- <https://endertech.com/blog/maintenance-bug-fixing-4-types-maintenance>
- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
- <https://www.nngroup.com/articles/ten-usability-heuristics/>
- <https://guides.github.com/introduction/flow/>
- <https://git-scm.com/>
- <http://mongoosejs.com/>
- <https://www.mongodb.com/>
- <https://quickleft.com/blog/six-reasons-we-split-front-end-and-back-end-code-into-two-git-repositories-working-with-github-repositories/>
- <https://en.wikipedia.org/wiki/FURPS>
- <https://spoonacular.com/food-api/>

Appendix

Use cases

UC 1: Sign up - Brief

Primary Actor: User

Main Success Scenario:

1. The user opens the webpage.
2. The system prompts the required information that needs to be filled out in order to sign up.
3. The user inputs username, email and password.
4. The user chooses the option to create a new account.
5. The system created a new user account and prompts the user to login.

UC 2: Login - Brief

Primary Actor: User

Main Success Scenario:

1. The user opens the webpage.
2. The user chooses the option to login.
3. The user is redirected to the login page.
4. The user inputs credentials: username or email and password.
5. The user chooses to login.
6. The user is now logged in and redirected to the search view.

UC 3: Logout - Brief

Primary Actor: User

Main Success Scenario:

1. The user logs into their account.
2. The system displays the home page.
3. The user selects to view their profile.
4. The system displays the user profile.
5. The user chooses the option to logout.
6. The user account is now logged out and is redirected to the landing page.