AppIchetto: an expense sharing application

Giuseppe Palazzolo
giuseppe.palazzolo@stud.unifi.it

Federico Vaccaro
federico.vaccaro@stud.unifi.it

Abstract

In this project we implemented a web application allowing the users to import long receipts from external sources (pdf documents or images), to split the expenses between friends, keeping track of the debts and ultimately to allow the users to pay the debts. We developed our application following the HCI principles, with a large focus on smartphone devices: we found these more suitable for letting the user to quickly accomplishing the given tasks, and for designing the interface through well-known elements.

1. Introduction

The development of our application has been divided in three core phases:

1. Needfinding [7]: we studied the needs of our users that had to be fulfilled, and then developed some *Personas* [8] who could represent our archetypal users;
2. Design of the functionalities: of the UI and then the actual implementation of the software;
3. Usability testing: we distributed the final application and assessed the quality of the user experience.

2. Needfinding

The need for an application that would help us in the managing of shared expense, has its roots in our daily experiences of off-site students, f.e. when sharing the costs of bills, grocery stores expense, or anything else. Thus, we wanted to directly provide a solution to our problems. Also, this means that we had the chance to directly observe how our users (and us) acted to cope with the expense sharing, where they struggled and how a software application could improve our quality of life. In each cases, there was one person buying together more items (the most common cases was after going at the grocery shop), so there is a debt from the participants people and to the buyer. The expense sharing issue evolves in different scenarios and difficulties for the various housemates:

- One person which spend the money buying some items with other friends, has to split the costs among the participant, which is boring and error-prone, especially when there are many objects to split;
- This "cost splitting" operation was often procrastinated for days or even weeks, yielding awkard situations between the participants about the expense (*With who did I shared this item? What is this item with this weird name in the receipt of two months ago?*);
- When someone was in a debt situation (meaning one owes something to another person) often would have liked to understand exactly how this debt was generated, and the buyer user often struggled to provide this information (*Did he pay me? Why I owe him 35 euros?*);

After this stage, we conducted some open interview (49 people partecipated, and we carefully avoided leading questions), using a Google Form to study the diffusion of our problems outside the walls of our houses and understanding the needs of the people related to the expense sharing. The demography of the participants were mostly people between 20 and 30 years old, living in different domestic situations, varying from our stereotype of off-site student. The participants lives with the family, with the partner, with some house-mates or even alone, so we managed to investigate far from our strict needs, but given the wide sampling of participants, we also expected someone not interested in our project.

In fact, we found from this poll that not everyone has our exact same problems with splitting items from long receipts, but almost everyone experience some shared purchases (Fig. 1) of various nature.

We found that most people use a manual method (Fig. 2) (pen and paper, with someone comfortable with spreadsheets), for doing the math and tracking these shared expense, and many people tends to get confused or even forget about tiny expenses after days. Overall, we found many people not satisfied with their current method of keeping track of their shared expenses, but also people who are fine with their current method.

Finally, we could identify mainly two Personas:

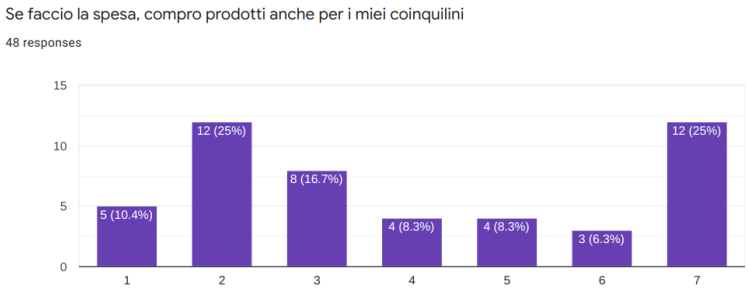

Figure 1. Results of the question about shared expenses.

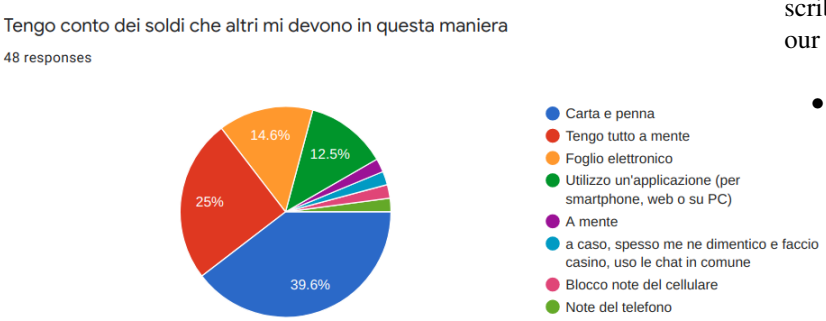

Figure 2. Results of the question about the method employed for keeping track of the friends small debts.

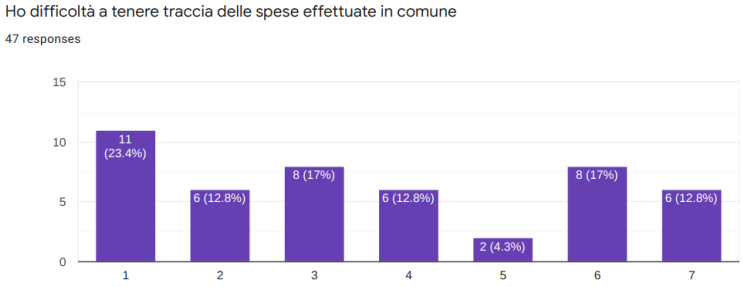

Figure 3. Results of the question about the method employed for keeping track of the friends small debts.

- **Francesco**: he is a University off-site student, far from his home. Since he loves cooking, he goes often to the grocery store and buys products for everyone. However, he has not an income, he can't offers each time the meal, so he has to split the expense with the others. After that he buys the food for several times consequently, he has to collect back the money. Also, it happens that his housemates forget why they owes money to Francesco, and he has issues when explaining how exaclty it is originated.

- **Emanuele**: he is a software programmer. He split some services with his friends. He lives by himself and he hates spending the little time he has at the end of the day for tracking his expense and notifying his friend one-by-one about the bills. Sometimes he also tends to be confusing with his current method, and forget if he gave the money to his friends, or if some of his friend paid for the services that is splitting with him.

3. Design & implementation

3.1. From the needs to the functionalities

Starting from the two personas and their context, we describe some typical scenarios which can happen between our users:

- **Francesco** is going to the grocery store for buying the ingredients for making the Sunday launch, when he has a little more time for cooking. He also will buy some items for the house-keeping who splits with the others, some personal other items for himself, and also one of his mates asks him if he can buy the cereals for breakfast. After he came back home, he has to calculate how much each friend owes him (which is more complex than dividing the total price from the receipt), so he spend for each receipt about 30 minutes doing the math and checking everything. Francesco's housemates also bought something for him, and would like to know if the debt is payed back with the new expenses.

- **Emanuele** is subscribed to services like Netflix and Spotify, which he shares with two group of friend. However, he's struggling to keep track of the monthly expenses: currently he's using a spreadsheet: while it helps, it still requires focus, so it is still error prone. Also, it happened that he had some discussion with a friend for being too much late with the payment, and would like to avoid discussions like these in the future.

Finally we identify what functionalities the application has to offer. In our context, the expense managing can be represented from two core aspects: **adding and sharing expenses** and **tracking the debt and credit** with a friend. More precisely, we identified the following functionalities:

- **Importing a receipt**: Some supermarket franchising provides a digital receipt available from their website in `pdf` format. The application should be capable of importing and parsing this document, sparing the time from the user of manually inserting one-by-one the items, in a such that the user has only to select, for each product, the participants. Of course, the user is allowed to manually insert the products he wants.

- **Costs computing**: the application should totally avoid the user from computing for each participant the debts originated from a receipt.
- **Debts & credits summary**: the user should be able to easily check the balance with a friend, and understand how it is originated. An user should be able to pay off his debt with granularity, or deleting the debt from a friend.
- **Social functions**: the user should be able to add new friend and also should be notified when there is some change about his status, *e.g.* when there is a new debt or when someone paid his debt.

3.2. Design of the application

After defining the functionality that the application has to fulfill, we started designing the User Interface and the software architecture. Our first choice was to develop the application as a **mobile device app**. The reasons behind are because a smartphone application is portable, so it is easier for the user to interface with it, it is easier to distribute respect to a desktop application, but most importantly we can develop a user interface such that the **user is already familiar with it** implying ease of use, thanks to the well-established **Material design** [4].

3.3. Structure of the application

We decided to structure the application in three macro-modules, accessible via "tabs" pattern: these modules are: **Status**, where the user can easily check the debts and credits with his friends and eventually proceeding for a more accurate inspection of the common tickets; **Import ticket**, where the tools for adding a new receipt into the system are provided; **Profile**, where the user can add and delete friend, and check the history of the tickets he bought and paid. Before implementing the different views, we first sketched by hand a mock-up of how we imagined and wanted the interface.

3.3.1 Import ticket view

In the logic of our application, the central entity is the **Ticket**. It represents a group of one or more products to share with the user's friends. Then, usually the first step is importing a ticket into the application, so we dedicated an entire tab for this activity. Before letting the user to add products into the ticket, some informations are required to the system. These are the **chosen market**, the **importing method**, and the **participant friends**; these informations are gathered via three subsequent steps (one view for step), and after completing each step, the user is guided to the next with an automated sliding. However, the user is free to slide back and forth, in case he want to correct some field. Following, the three steps are illustrated.

- **Select market**: as example we inserted three italian popular supermarket companies, this selection allows the system to employ the right `pdf` parsing algorithm (Fig. 4) or simply remembering where he bought the products; there is also a 'generic' option;

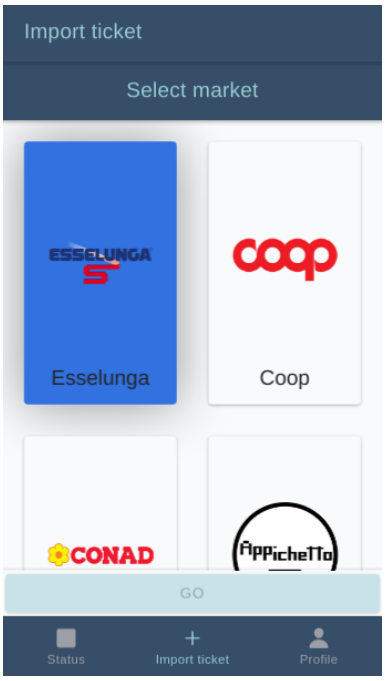

Figure 4. Select market.

- **Importing method**: it is possible to choice between: scan a photo and importing the ticket with OCR algorithms (experimental feature), letting the user to create the ticket manually, or importing the receipt by reading a `pdf` document (Fig. 5);
- **Select participants**: here the user select who are the friends participating to the ticket; the user (who is the one who spent money for the receipt) is included by default (Fig. 6).

After that the three steps are completed, the *Go* button is unlocked (with visual feedback) and the user can finally adding elements to the ticket (Fig. 7). Of course, in case of selecting the import by `pdf` method, the products are already in the ticket.

After the user clicks on the *Go* button, the view in Fig. 7 is shown. Here, the user can see the scanned ticket, or eventually he is able to add new product. The user can modify every product clicking on the *pencil* button: the relative product will change the background (for notifying that the

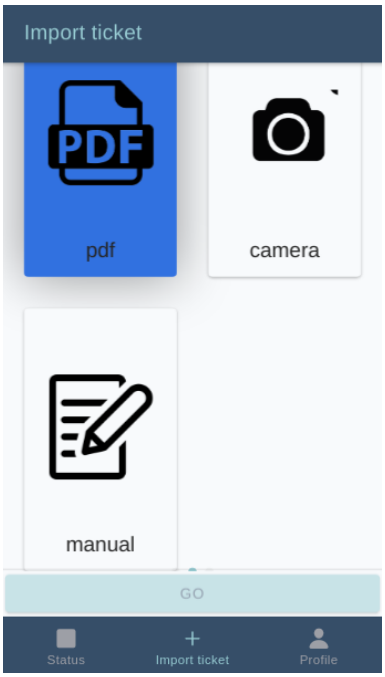

Figure 5. Select method.

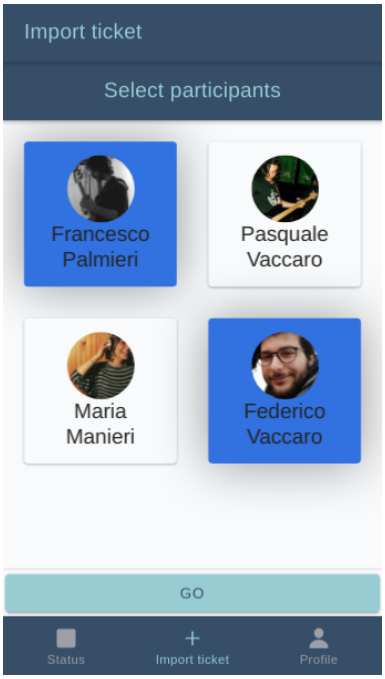

Figure 6. Select participants.

edit mode is active for the selected item) and will be editable, or removed clicking on the common *trash can* icon. When *person* button is clicked, the user is able to add participants to a product: then, a pop-up window is shown, and the person icon will be filled, if at least one friend is selected. This is useful because it is not allowed to save on

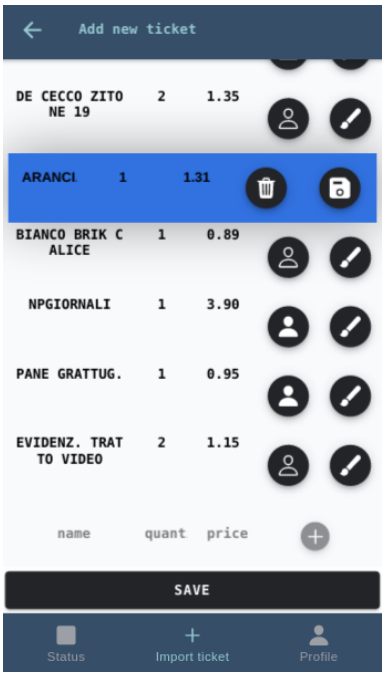

Figure 7. Products in the ticket.

the database a ticket in case there is one or more product with any participants, so the user easily understand if he missed to select the participants for an item. When saving the ticket, for each products will be automatically computed the price that every participant has to pay, considering quantity, price and number of participants. Finally, the ticket will be splitted and saved on the database, and every participant will be notified of the new ticket.

3.3.2 Status view

After adding some friends and sharing tickets, in Fig. 8 is how the Status view appears to the user.

From this view the user can access the basics informations he needs about his accountings, so here is shown a recap about each user owes/is owed to/from one of his friend, plus a button for the notification, showing to the users if there is something new. We designed this view following common design patterns and gestures in mobile applications: we employed a list of cards, communicating that the user can click on the various elements, and for each friend there is a circular avatar (as in many social applications). At the top-right position, there is a button that open a pop-up section where are displayed the content of the notification messages. With a left-slide gestures, is possible to delete the notification (Fig. 9).

By clicking on one friend, is opened a view where the user can review all the shared expenses (Fig 10).

There are two sections: **Debts** and **Credits**. In the first section there are the tickets bought from the friend with who

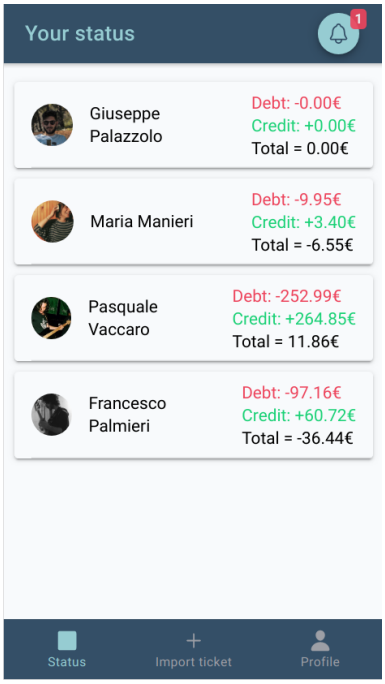

Figure 8. Status page view.

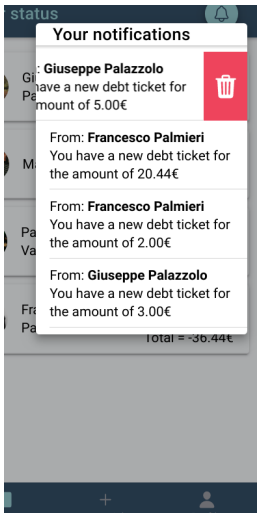

Figure 9. Notification menu.

some item have been shared. There is a card for each ticket, where the basics information about it are displayed: the timestamp of the ticket, and how much the users owes to the friend for that. There is also an arrow, for letting the user to understand he can expand and collapse the card for inspecting the content of the ticket. The **Credit** section is basically the same, but conversely contains the items bought by the user. For navigating back and forth between Debts and Credits section, is also employed the commong gesture of *swiping*.

In the footer bar, the debts and credits are summarized

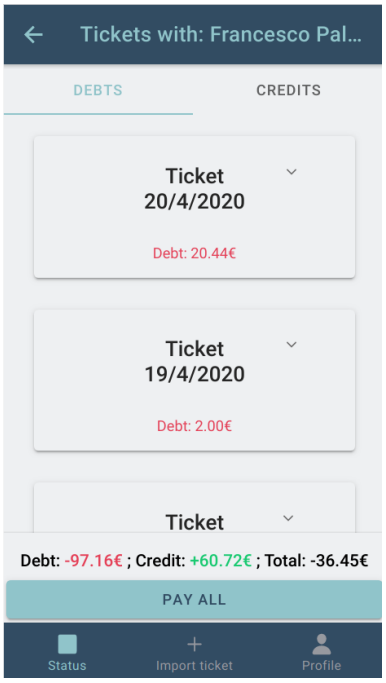

Figure 10. Inspecting the shared ticket with a user.

again, and there is a button (*Pay All*, Fig. 11) which is active only if the user has a debt with the selected friend. This will wipe each ticket between the user and his friend. Also, the *Pay Ticket* and *Mark as paid* buttons are available (relatively in the Debt and Credit sections) after inspecting a ticket, allowing the user to have more control on the single tickets. After clicking on the button, is still possible to abort the payment, for avoiding that the user commits unwanted changes to the status with his friend.

3.3.3 Profile view

In the Profile view, the user can performs generic actions such as log-out from the application, switch between light/dark themes, add or delete friends, inspect old tickets. (Fig. 12).

Friends section show the actual friends with the with the possibility of add new friend clicking in *Add friend* button and inserting his email or remove it swiping to left and clicking the *trash* button (Fig. 13).

The **Paid ticket** section shows the tickets already paid to a friend, the user can click the ticket if wants to see the products details Similarly, in the **My ticket** section the user can review his part of the tickets he imported.

The last section is **Ticket history**, that shows all the ticked that the user imported here he can see what he bought when and with whom he shared it.

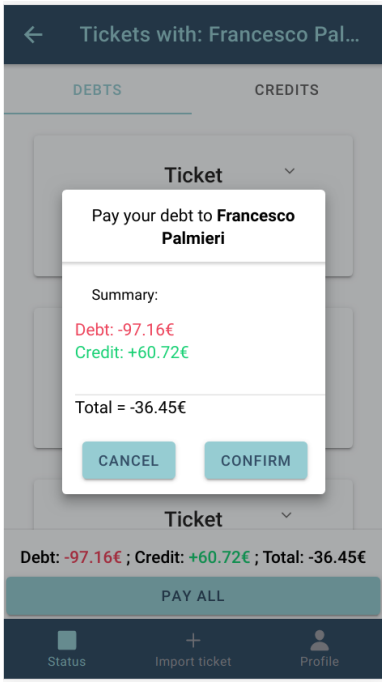

Figure 11. Payment pop-up.

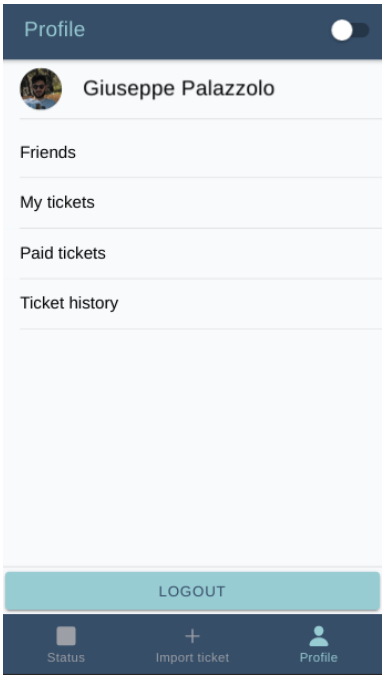

Figure 12. Profile view.

3.4. Implementation

For the implementation of this application, we relied on three popular TypeScript tools:

- **Ionic V5** [1]: open source framework for building the easily UI with components based on the Material design [4];
- **Angular** [3]: core framework base on TypeScript for developing the front-end side of web applications;
- **Firebase** [2]: NoSQL cloud database for storing the models;

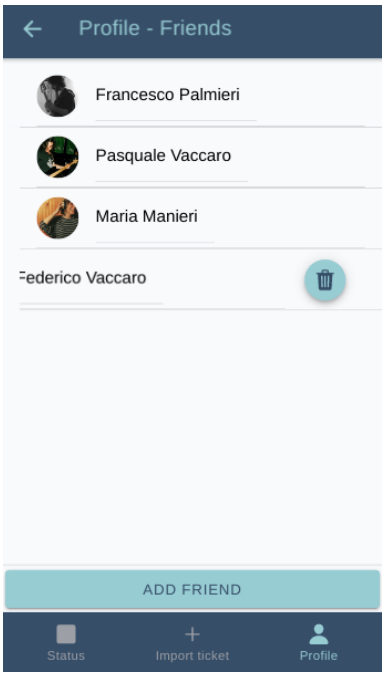

Figure 13. Friends list.

In particular, most parts of the app are based on the **Model-View-Controller** (Fig. 14) [9] architectural pattern, allowing us to build the system in a modular and hierarchical manner, so we could work in parallel without interfering with each other, flavouring also the maintainability of the code according to the principles of the OOP. Also, thanks to the **RxJS** library we managed to synchronize the application with the changes of the database in real-time. Firebase was an easy choice for several reasons: its basic features are free, it is well-integrated with the Angular environment, since it is a document-oriented [5] [6] real-time database, it fits the nature of our problem more then a more traditional relational DB. Also, thanks to Firebase we could easily bind the login of our application to the Google Login provider. We found this essential, because pratically it is not needed any registration, given that lots of people already have a Google account, thus making the application more accessible.

4. Usability Test

After developing the application, we let some of our friend (who we thanks for participating!) testing it, after the

Table 1. Results of the SEQ questions after the Usability Tests

N	Question	μ	σ
1	The application is visually pleasant	6.07	0.95
2	The meaning of the buttons and the elements of a page is clear	6.15	0.80
3	The application results smooth and fluid	6.07	0.75
4	The application results hard to use	1.92	0.64
5	Inserting a new ticket is challanging	1.46	0.77
6	It is easy to understand the functionalies of the application	6.23	0.83
7	It is hard to understand the origin of a debt from a friend	1.69	0.63
8	The positioning of the buttons is wrong and illogical	1.61	0.87
9	Overall, the system is enjoyable	6.38	0.76
10	The application has every functionality I need	6.00	1.08
11	I learned to use better the application after a while	4.69	1.10
12	I ignored the notifications	1.76	1.23
13	It is easy to add or remove friends	6.53	0.66
14	It is easy to pay a debt	6.46	0.77
15	I am overall satisfied	6.15	0.68

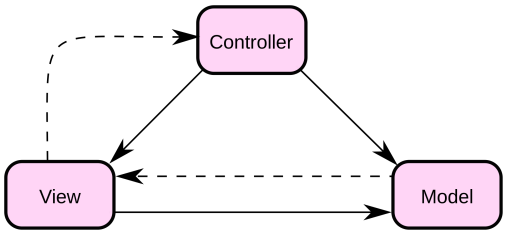

Figure 14. MVC pattern schema.

distribution of the `apk`, or simply via browser emulating via mouse the smartphone interaction. We asked to perform 7 different actions for testing each feature available from the application. They tested the application under our supervision (the tests were mostly conducted via web-conference, we asked to stream their desktop), not just for helping them, in case there was something more difficult than expected, but also for directly observing how much the interface felt natural and their overall experience. These were the steps we asked to follow:

1. Add a friend
2. Add a new receipt
3. Add/update/delete elements in the receipt
4. Check the balance with a friend
5. Inspect the tickets shared with a friend
6. Make the payment of a debt to a friend
7. Inspect the paid tickets

For a better observation of how they reacted to our application, we limited the group of testers to two people at one time. In case one of the testers had some difficulty during one of these steps, we asked to repeat the steps $2-6$ one more time, thus letting him to take confidence with the interface, and understanding how the experience and the ease of use improved after a while. Finally they replied to 15 SEQ questions, consisting of a discrete number spanning from 1, if the tester strongly disagree with the statement, to 7 otherwise; also, we left one last optional open-answer question for a free thought or suggestion about the application.

We asked to adding new receipts to share between the people who tested or between us, for staging a real scenario about expenses (a dinner to a friend's home), and providing them real `pdf` receipts. In Tab. 1 the results of the tests are shown. Our testers were overall satisfied with the application and easily understood how to complete the various tasks without our direct intervention, and found the workflow of the application smooth. The application was not perfect, though the few criticalities they found were mostly about minor implementation bugs (*e.g.* a *back* button disappeared, so one user were struggling to find a section of the application) which we couldn't spot in our tests.

5. Conclusion & further developments

Designing and developing an application about costs splitting between friends, has been for us a challanging task, both for the back-end and the front-end, but in this project we focused on developing an interface that is self-explaining and fluid. Thanks to the phases and principles of the Humen-Computer Interaction (needfinding, development and usability testing) we managed to produce an user interface that felt natural and friendly. Our testers were after all satisfied with the application, as the results of the tests tell. Most importantly, we also began using the application for finally solving our initial domestic problems! Also, we have in mind some ideas to further extend the functionali-

ties of the application, and moreover we received from our testers some helpful suggestions.

- Integrate a payment system (e.g. PayPal): we noted a certain attitude between the people who helped us during the needfinding for using electronic payment systems, which also we believe would be handy;
- Inserting a ”group” feature: one of the testers suggested us to create a network or group of friend, where each ”circular debt” would be automatically solved. However, this feature implies that would not be possible to track the origin of the debt (or credit), which was a central point to us, for transparency;
- Push notifications in addition to the current notification system;
- Improving the OCR algorithm for scanninc receipts.

References

[1] Drifty. Ionic V5 Website. `https://ionicframework.com/blog/`, 2020. [Online; accessed in may 2020].

[2] Google. Firebase Website. `https://firebase.google.com/`, 2020. [Online; accessed in may 2020].

[3] Google. Google Angular Website. `https://angular.io/`, 2020. [Online; accessed in may 2020].

[4] Google. Material: a design system backed by open-source code that helps teams build high-quality digital experiences. . `https://material.io/design`, 2020. [Online; accessed in may 2020].

[5] A. B. M. Moniruzzaman and S. A. Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191, 2013.

[6] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, and S. Belfkih. Comparison and classication of nosql databases for big data. 05 2015.

[7] D. Patnaik and R. Becker. Needfinding: The why and how of uncovering people's needs. *Design Management Journal (Former Series)*, 10:37 – 43, 06 2010.

[8] J. Pruitt and J. Grudin. Personas: Practice and theory. In *Proceedings of the 2003 Conference on Designing for User Experiences*, DUX 03, page 115, New York, NY, USA, 2003. Association for Computing Machinery.

[9] Wikipedia. MVC pattern - Wikipedia. `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`, 2020. [Online; accessed in may 2020].