Arab Open University- Saudi Arabia

Faculty of Computer Studies

Tools Rental Platform

Laith Hussam Shono 101708516

TM471: Final Year Project, 2021

Supervisor: Dr. Basil Kasasbeh

Abstract

This project aims to create an e-commerce online rental platform. The project aims to provide a platform where users can offer their tools for rent and to rent other users' offered tools. Another aim is to provide multiple features protecting the users' tools and protecting them from scams.

The system is called "Rentool" and its front-end is three apps, an Android app, an iOS app, and a web app hosted on rentool.site. All these apps are built using Flutter. The system's backend is built with Firebase. Firebase Authentication for users' account and authentication, Firestore for the database, Cloud Functions for running backend code, Cloud Storage for storing pictures, videos, and other files, Firebase Hosting for hosting the web app, Firebase Cloud messaging for sending notifications to the owner. And Algolia for Searching.

The system will handle payments between renters and owners and the system's payment solution is implemented using *Checkout*.

Acknowledgements

I would like to thank my family and friends for all the support. I also thank Dr. Basil for all his help with this course and for answering my questions about writing the report.

Also, big thanks to the Stack Overflow community for helping me fix the bugs I encountered while building this project.

Table of Contents

1. Introduction

The internet helped change the way we live. It allowed information to be available to everyone. Including information on how to do specific jobs, some of which we always hired a professional to do. This resulted in a lot of people starting doing these jobs themselves. According to (National Association of REALTORS, 2019), 47% of homeowners do projects on their homes by themselves. However, some of these jobs require specific tools that not everyone may have. Some people may go and buy such tools, just to end up using them once and never again. Which results in a loss of money, and the tools taking space and cluttering up the place.

A solution is to rent the tools instead of buying them. However, this may be harder than it sounds, as it's not easy to find a place to rent specialized tools, and there are not many rental platforms online. And most of the ones available are businesses-to-consumer websites, which don't allow people to rent their own tools. The few ones that are peer-to-peer either lack major features or are not available for everyone.

The solution is a peer-to-peer e-commerce rental platform. Allowing the owners to offer their tools for rent and make a profit out of them. And allowing others to request to rent other people's tools and avoid buying a new one. The platforms also provide a unified place where people can look for tools to rent, instead of searching for hours across multiple sites online.

In this report, I will first go through my research on the subject and the methods I found and will be following. Then, I will go through the domain modeling process, identifying the use cases and their textual description. Then I will list the functional and non-functional requirements. And finally, I will list the system constraints and operations. And a glossary of all the important terms will be at the end of chapter 3.

Chapter 2 – Literature Review

When I chose this subject, I was mostly concerned with people stealing or damaging the tools. So, I searched for methods people use to prevent that. I also searched for current platforms that people use and learned what works best and what doesn't.

First, I started searching for platforms and methods of renting tools online. I found most of them were business-to-consumer (B2C), meaning they were businesses offering their own tools for rent/hire. Where you select a tool, check for availability in one of their stores, and pick it up from there. Two of these businesses were HSS Hire and United Rentals.

I found an app that did offer good peer-to-peer (P2P) tools renting services. The app is Sparetoolz. However, the app has a really small user base, thus there are almost no tools offered. And that the app is only available on the Apple App Store and planned to come soon to Google Play, With no announced plans for the web. So, with all of these problems, I couldn't get much from Sparetoolz.

So, I searched for general renting platforms (not specific for tools). I found the most widely used in the app store were Rent4Me, MyRent, and Rent iT. However, MyRent and Rent iT are only available in the United States. But, Rent4Me is available globally, it's on the top of the rent apps chart, and it's available on Android, iOS, and the web. So, I used it the most to compare and research the subject.

While Rent4Me seemed perfect at first. I found some of the methods it uses are limiting the system. Offering something, for example, could take a while as you need the administrators to accept the post first, then it can be published and available for others. The same thing for IDs, they must be approved by the administrators before the user is verified. While these methods seem better and more secure at first, they could push away the users especially when the user base gets bigger and moderation starts taking longer, and it also delays the user experience. Instead, it's better to let the users evaluate the posts and users' IDs themselves while giving them instructions and tips on how to do so safely.

Another concerning choice with Rent4Me is that an unverified user (a user who didn't provide an ID or phone number) can still offer and request stuff, and this is a security concern for a lot of users.

In addition, I found multiple bugs and problems with the system, here are some of the most troubling and important to avoid ones:

- phone validation doesn't work. I tried on my personal phone number and I tried an online USA number. In both cases, the validation message doesn't arrive. Therefore, I couldn't get my account fully verified.
- The user interface is confusing to navigate.
- The search feature is a bit buggy, especially when zooming out in the map view.
- Sometimes loading pages become so slow (more than 5 seconds).
- Uploading pictures is buggy where sometimes you have to upload them multiple times to work.

- Lack of feedback after user interactions. For example: when uploading IDs, it won't give feedback that the image was uploaded, and you need to refresh the page to see it

Another method of renting I found lots of people using is renting through social media platforms such as Instagram, Twitter, Facebook…etc. Where they post about their tools and receive requests in direct messages. Then paying in-cash on delivery.

Then, I started looking for the best methods of the renting process. I found the best and most used method is for the owner to set an insurance amount/deposit that will be held on the renter's credit card and will be returned once the tool is returned in a good condition.

Another method is the renter and owner providing their IDs so they are verified and in case of any problems they can take the problem to the police or court. This is also helpful on the system side, as it makes banning a user from the system much easier as everyone has only one ID number.

2.1 Payments

The system should be able to send payments from and to users. Therefore, it must contact a payment gateway to handle these payments. According to the (Unified National Platform, 2021), the approved e-payment gateways in Saudi Arabia are *Moyasar*, *Paytabs*, *Hyperpay*, and *Amazon payment services* (*Payfort* previously). After checking their documentation and contacting them I found that *Moyasar*, *Paytabs*, and *Hyperpay* don't support saving users credit/debit cards, which is a necessary feature for this system since it's not PCI-certified. As for *Amazon payment services*, they didn't reply to my question nor my request for an account.

Outside of the gateways approved in Saudi Arabia, *Stripe* was another popular and well-supported option worldwide with a lot of libraries and resources online. However, although *Stripe* was great for receiving payments from users, sending payments was much harder as it requires the user to create a *Stripe* personal account, and *Stripe* is not available in Saudi Arabia yet.

After more research I found *Checkout*, and it supported merchants accounts in Saudi Arabia. However, the most useful feature was *Card Payouts* Which allows the system to payout (send payment) to a user's credit/debit card directly (without the user having to create a *Checkout* personal account) instead of having the user enter his/her bank details. After further reading *Checkout*'s documentation, I found it has all the features needed and I decided it was the best option for this system.

Chapter 3 – Requirements and analysis

In this chapter, I'll show the first 3 phases of the development which are domain modelling, requirements gathering, and analysis.

3.1 Domain modelling:

3.1.1 Business use cases:

1. Offer tool
2. Edit/remove tool post
3. Accept a tool-request
4. Deliver tool
5. Request a Tool
6. Edit/remove tool request
7. Return tool
8. Create an account
9. Log into account
10. Add/edit/remove user review
11. Ban a user

3.1.2 use case diagram:

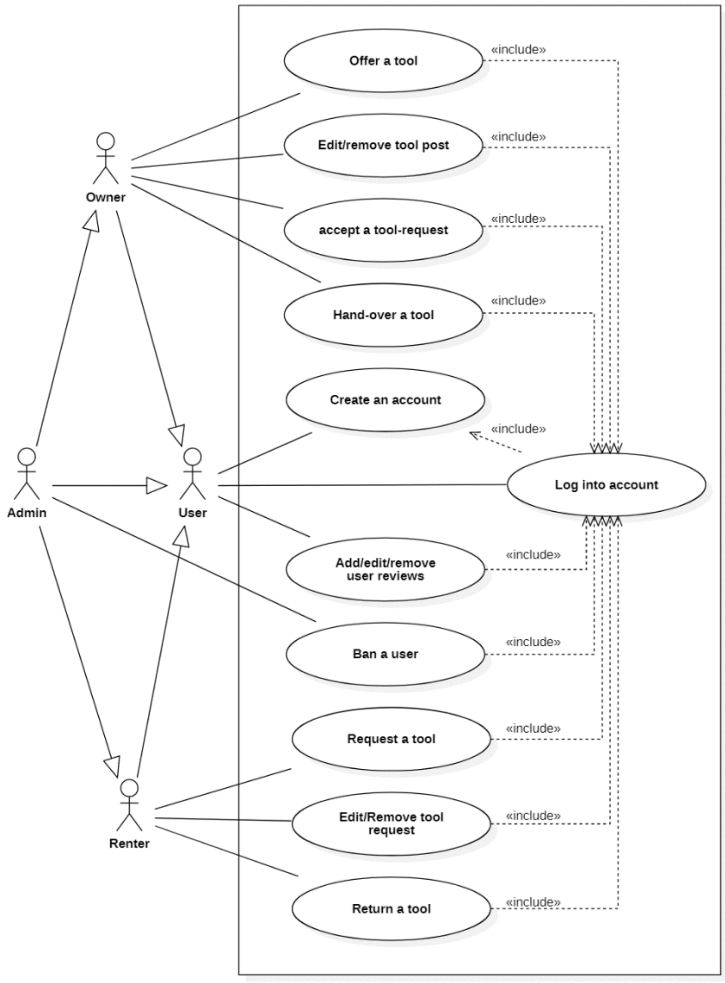

Figure 1- Use case diagram

3.1.3 Textual description of the use cases:

ID: UC1

Name: Offer a tool

Initiator: Owner

Goal: creating a post for a tool.

Precondition:

1. The owner must be logged in and provided his/her ID number.
2. The owner must have provided his/her credit card details.

Postcondition: A post will be created where others can request to rent the tool.

Main success scenario:

1. The owner requests to create a new post.
2. The system ensures the owner is authorized for posting.
3. The owner enters the post details.
4. The owner presses the confirm button.
5. The system ensures that all inputs are valid.
6. The system creates a post in the database.
7. The system directs the owner to the post page.

Extensions:

2.1.1. The owner isn't authorized to post.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the user.

5.1.1. Some fields are not valid.
5.1.2. The request will not be created.
5.1.3. The system will inform the user of the invalid values.
5.1.4. Continue at step 3.

ID: UC2 - A

Name: Edit tool post.

Initiator: Owner.

Goal: Editing a post of a tool.

Precondition:

1. The post must exist.
2. The owner must be authorized to edit the post.

Postcondition: The tool's post will be changed as the owner specifies.

Main success scenario:

1. The owner requests to edit a post.
2. The system ensures the owner is authorized to edit the post.
3. The owner edits the desired post details.

4. The owner confirms the changes.
5. The system ensures that all inputs are valid.
6. The system changes the post details in the database.

Extensions:

2.1.1. The owner isn't authorized to edit the post.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the owner.

5.1.1. Some fields are not valid.
5.1.2. The request will not be created.
5.1.3. The system will inform the user of the invalid values.
5.1.4. Continue at step 3.

ID: UC2 - B

Name: Remove tool post

Initiator: Owner

Goal: Deleting a post of a tool.

Precondition:

1. The post must exist.
2. The owner must be authorized to remove the post.

Postcondition: The post will be deleted and removed from the database. And users won't be able to access it anymore.

Main success scenario:

1. The owner requests to remove a post.
2. The system ensures the owner is authorized to remove the post.
3. The owner confirms that (s)he wants to delete the post.
4. The system deletes any requests to the post.
5. The system deletes the post from the database.

Extensions:

2.1.1. The owner isn't authorized to delete the post.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the owner.

ID: UC3

Name: Accept request

Initiator: Owner

Goal: Accepting a renter's request to rent the owner's tool.

Precondition:

1. A request must have been sent from a renter on a post.

2. The owner must be authorized to accept/reject requests for the post
3. The tool must not have an accepted request.

Postcondition: The request will be accepted, and a chat room will open for the owner and renter.

Main success scenario:

1. The owner requests to see the requests for the post.
2. The system ensures the owner is authorized to view and accept requests for the post.
3. The owner accepts the renter's request.
4. The system opens a chat room between the renter and owner.
5. The system sends a notification for the renter.

Extensions:

3.1.1. The owner isn't authorized.
3.1.2. The request will be denied.
3.1.3. The system displays a prompt for the owner.

3.1.1. The owner rejects the renter's request.
3.1.2. The system deletes the request.

ID: UC4

Name: Deliver tool

Initiator: Owner

Goal: Handing over the tool to the renter and start the rent period.

Precondition:

1. The owner and renter agreed to meet at a place.

Postcondition: the renter will get the tool and the rent starts.

Main success scenario:

1. The owner and renter confirm their arrival at the meeting place and that they met.
2. The owner and renter take pictures and/or videos to document the state of the tool before the hand-over.
3. The owner and renter upload the pictures and/or videos to the system.
4. The owner and renter confirm the pictures and/or videos.
5. The system gives the owner the renter's ID number and gives the renter the owner's ID number.
6. The owner and renter ask for each other's ID and confirm they match the ones in the system.
7. The system deducts the insurance price and the intended rent from the renter's credit card.
8. The system pays the owner the rent price.
9. The owner gives the tool to the renter.
10. The system updates the database and sets the rent start time.

Extensions:

1.1.1. Either the owner or renter didn't confirm arrival and meeting
1.1.2. The delivery will not continue.

4.1.1. Either the owner or renter doesn't agree on the pictures and/or videos.
4.1.2. The owner and renter try again and go back to step 2.
4.1.3. If they didn't agree: the owner/renter cancels the delivery and tool-request.
4.1.4. The system deletes the tool-request.

5.1.1. The owner or renter refuses to give their IDs.
5.1.2. The owner or renter cancels the delivery and tool-request.
5.1.3. The system deletes the tool-request.

7.1.1. The system fails to deduct the money
7.1.2. The operation gets canceled

8.1.1. The system fails to pay the owner
8.1.2. The operation gets canceled

ID: UC5

Name: Request tool

Initiator: Renter

Goal: Sending a request to rent a tool on a certain post.

Precondition:

1. The renter must be logged in and provided his/her ID number.
2. The renter must have provided his/her credit card details.
3. The renter must not have a previous request for the tool (i.e., the renter can only have one request for a tool).

Postcondition: a request will be sent to the owner on the tool post, where the owner could decide to accept or reject it.

Main success scenario:

1. The renter requests to send a *request* to a post.
2. The system ensures the renter is authorized for requesting.
3. The system ensures the renter doesn't have a previous request on the post.
4. The renter enters the request details.
5. The renter confirms the request.
6. The system ensures all inputs are valid.
7. The system updates the database and adds the request.
8. The system sends a notification to the owner.

Extensions:

3.1.1. The renter isn't authorized.
3.1.2. The request will be denied.
3.1.3. The system displays a prompt for the renter.

3.1.4. The system directs the user to a screen to enter his/her credit card details.

3.1.1. The renter has a previous request on the post.
3.1.2. The request will be denied.
3.1.3. The renter will be redirected to the previous page.

8.1.1. Some fields are not valid.
8.1.2. The tool-request will not be created.
8.1.3. The system will inform the renter of the invalid values.
8.1.4. Continue at step 4.

ID: UC6-A

Name: Edit a request

Initiator: Renter

Goal: Editing the details of a request.

Precondition: The renter must be authorized to edit the request.

Postcondition: The request details will be changed as the renter specifies.

Main success scenario:

1. The renter requests to edit a *request*.
2. The system ensures the renter is authorized to edit the request.
3. The renter enters the changes (s)he desires.
4. The renter confirms the changes.
5. The system ensures all inputs are valid.
6. The system updates the request in the database.

Extensions:

2.1.1. The renter isn't authorized.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the renter.

5.1.1. Some fields are not valid.
5.1.2. The edit will be denied.
5.1.3. The system will inform the renter of the invalid values.
5.1.4. Continue at step 3.

ID: UC6-B

Name: Remove a request

Initiator: Renter

Goal: Removing/deleting a request.

Precondition: The renter must be authorized to remove the request.

Postcondition: The request will be removed from the database, and becomes unavailable to the owner.

Main success scenario:

1. The renter requests to remove a *request*.
2. The system ensures the renter is authorized to remove the request.
3. The renter confirms the decision.
4. The system removes the request from the database.

Extensions:

2.1.1. The renter isn't authorized.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the renter.

ID: UC7

Name: Return a tool

Initiator: Renter

Goal: Handing back the tool to the owner

Precondition:

1- The owner and renter agreed to meet at a place.

Postcondition: The tool is back with the owner and the owner can accept other requests.

Main success scenario:

1. The owner and renter confirm their arrival at the meeting place.
2. The owner confirms the state of the tool is unchanged.
3. The renter returns the tool.
4. The system calculates the remaining of the insurance money.
5. The system returns the remaining of the insurance money to the renter.
6. The system sends the deducted money to the owner
7. The system ends the rent.

Extensions:

1.1.1. The owner or renter doesn't confirm arrival and meeting.
1.1.2. If the rent period has ended, the user has 1 day to deliver the tool to the authorities and asks for a paper to confirm the delivery.
1.1.3. The renter uploads the paper.
1.1.4. The admin reviews the paper and confirms it.
1.1.5. The system sends a notification for the owner of the tool position.
1.1.6. If the renter didn't deliver the tool each day's rent will be deducted from the insurance money until the renter returns the tool or until the insurance money is all deducted.

2.1.1. The owner claims the tool is damaged.
2.1.2. The renter admits it
2.1.3. The renter and owner agree on a compensation price.

2.1.4. Continue at step 5.

2.2.1. The owner claims the tool is damaged.
2.2.2. The renter refuses the claims.
2.2.3. The owner and renter take pictures and/or videos to show the state of the tool.
2.2.4. The owner and renter upload their pictures and/or videos to the system.
2.2.5. The system creates a disagreement case.
2.2.6. The admin reviews the case and arrives to a decision.
2.2.7. The system sends a notification to the renter and owner informing them of the decision.
2.2.8. If the tool is indeed damaged, continue at step 2.1.3
2.2.9. If it's not damaged, the owner and renter agree on another meeting.
2.2.10. Continue at step 1 and skip step 2.

ID: UC8 - A

Name: Create an account using an email address.

Initiator: User

Goal: Creates an account for the user to log into.

Precondition:

1- The user must have an email address.
2- The user must have an ID number.
3- The email address and ID number must not be on the banned list.

Postcondition: an account is created for the user to log into, where (s)he can rent or offer tools.

Main success scenario:

1- The user requests to create a new account.
2- The user enters the name, email address, ID number, and password.
3- The system ensures all inputs are valid.
4- The system ensures the user isn't on the banned list.
5- The user enters his/her credit card details.
6- The system creates a new user account with the provided details.

Extensions:

3.1.1. Some fields are not valid.
3.1.2. The request will be denied.
3.1.3. The system will inform the renter of the invalid values.
3.1.4. Continue at step 2.

4.1.1. The user is on the banned list.
4.1.2. The request will be denied and the account will not be created.
4.1.3. The system will display a prompt for the user.
4.1.4. The system will redirect the user to the homepage.

ID: UC8 - B

Name: Create an account using Google, Facebook, Microsoft, or Apple account.

Initiator: User

Goal: Creates an account for the user to log into.

Precondition:

1. The user must have a Google, Facebook, Microsoft, or Apple account.
2. The user must have an ID number.
3. The email address and ID number must not be on the banned list.

Postcondition: an account is created for the user to log into, where (s)he can rent or offer tools.

Main success scenario:

1- The user requests to create a new account.
2- The user signs in using one of the sign-in buttons for Google, Facebook, Microsoft, and Apple.
3- The user logs in to the selected account.
4- The user enters the ID number.
5- The system ensures the user isn't on the banned list.
6- The user enters his/her credit card details.
7- The system creates a new user account with the provided details.

Extensions:

5.1.1. The user is on the banned list.
5.1.2. The request will be denied, and the account will not be created.
5.1.3. The system will display a prompt for the user.
5.1.4. The system will redirect the user to the homepage.

ID: UC9 - A

Name: Log into account using an email address.

Initiator: User

Goal: Log the user into his/her account where (s)he can offer and/or rent tools.

Precondition:

1- The user must have previously created an account.
2- The user account must not be banned.

Postcondition: the user is logged into his/her account, and (s)he can start renting or offering tools.

Main success scenario:

1- The user requests to log into his/her account.
2- The system asks for the email address and the password.
3- The user enters the email address and the password.
4- The system checks if the email address and the password are correct.

5- The system ensures the user is not banned.
6- The system logs the user into the account.

Extensions:

4.1.1. the email address and the password are not correct.
4.1.2. The user will not be logged in.
4.1.3. The system will inform the renter.
4.1.4. Continue at step 2.

5.1.1. The user is on the banned list.
5.1.2. The request will be denied, and the user will not be logged in.
5.1.3. The system will display a prompt for the user.
5.1.4. The system will redirect the user to the homepage.

ID: UC9 - B

Name: Log into account using Google, Facebook, Microsoft, or Apple account.

Initiator: User

Goal: Log the user into his/her account where (s)he can offer and/or rent tools.

Precondition:

1- The user must have previously created an account.
2- The user account must not be banned.

Postcondition: the user is logged into his/her account, and (s)he can start renting or offering tools.

Main success scenario:

1- The user requests to log into his/her account.
2- The user logs in using one of the sign-in buttons for Google, Facebook, Microsoft, and Apple.
3- The user logs in the selected account.
4- The system ensures the user is not banned.
5- The system logs the user into the account.

Extensions:

5.1.5. The user is on the banned list.
5.1.6. The request will be denied, and the account will not be created.
5.1.7. The system will display a prompt for the user.
5.1.8. The system will redirect the user to the homepage.

ID: UC10 - A

Name: Add/edit user rating.

Initiator: Rating-user

Goal: Adds or edit a review of a target-user.

Precondition:

1. The rating-user and target-user must have a previous transaction to create a new review.
2. The rating-user must be authorized to edit the review.

Postcondition: Adds/edits a review to the rating of the target-user.

Main success scenario:

1. The rating-user requests to rate the target-user.
2. The system ensures the precondition.
3. The rating-user enters the details.
4. The rating-user confirms the changes.
5. The system ensures all inputs are valid.
6. The system creates/edits the review.

Extensions:

2.1.1. The rating-user and target-user didn't have a previous transaction, or the rating-user is not authorized to edit the review.
2.1.2. The request will be denied.
2.1.3. The system displays a prompt for the renter.

5.1.1. Some fields are not valid.
5.1.2. The request will be denied, and the review won't be created/edited.
5.1.3. The system will inform the renter of the invalid values.
5.1.4. Continue at step 3.

ID: UC10 - B

Name: Remove user rating.

Initiator: Rating-user

Goal: Deletes the rating-user's review of the target-user.

Precondition: The Rating-user must be authorized to delete the review.

Postcondition: The rating-user's review of the target-user will be deleted, and target-user rating will be updated.

Main success scenario:

1- The rating-user asks for the review to be deleted.
2- The rating-user confirms his/her choice.
3- The system ensures the precondition.
4- The system deletes the review.
5- The system updates the target-user rating.

Extensions:

3.1.1. The rating-user isn't authorized.
3.1.2. The request will be denied.
3.1.3. The system displays a prompt for the renter.

ID: UC11

Name: Ban user

Initiator: Admin

Goal: Permanently ban a user from the platform.

Precondition:

Postcondition: the user will be permanently banned from the platform, using their email address, and ID number if provided.

Main success scenario:

1- The admin requests to ban a user.
2- The admin enters the ban details.
3- The admin confirms his/her decision.
4- The system will add the user email address and ID number (if provided), to the banned-list.

3.1.4 Activity diagrams

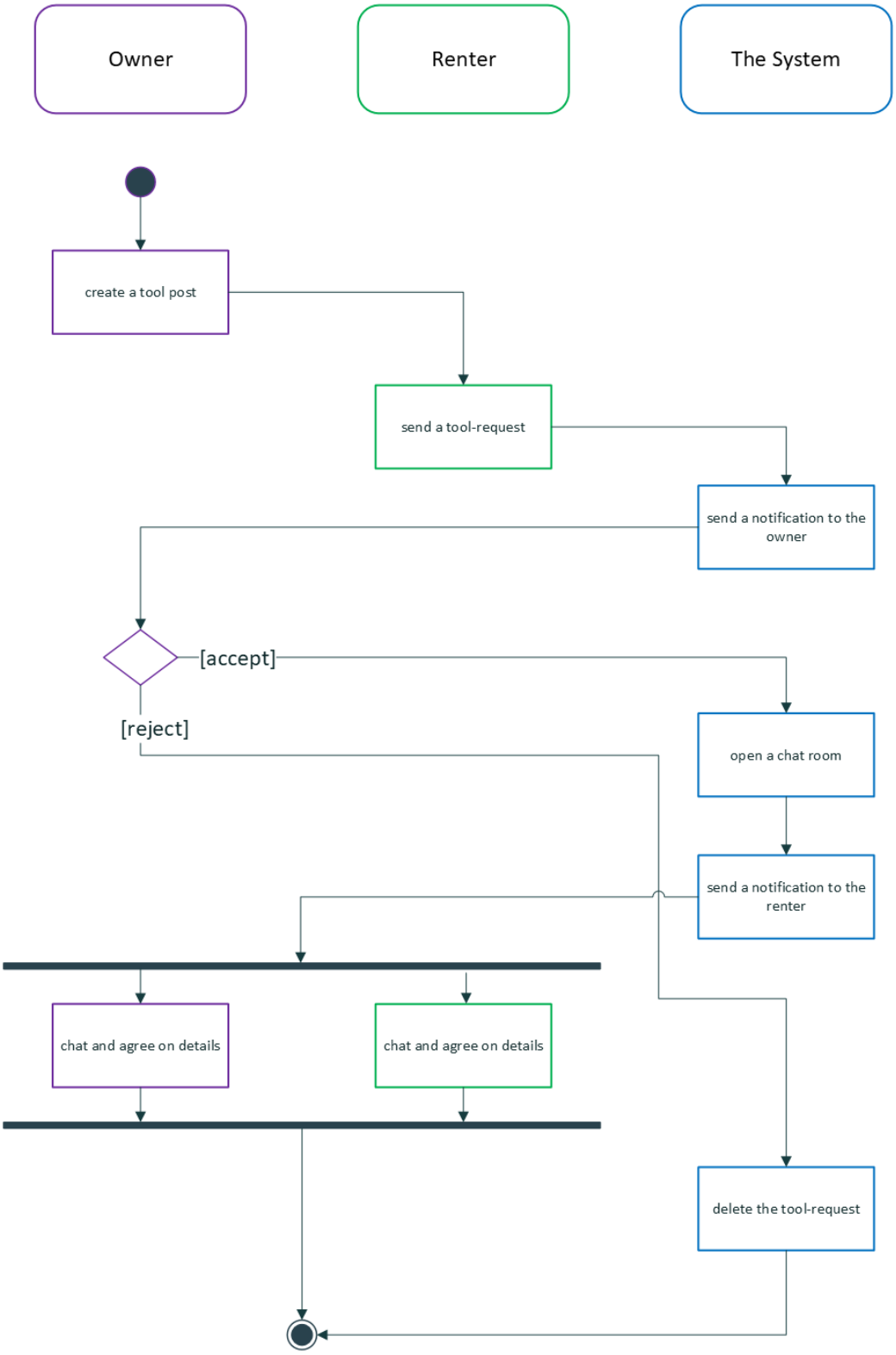

Figure 2 - Activity diagram of accepting/rejecting tool-requests

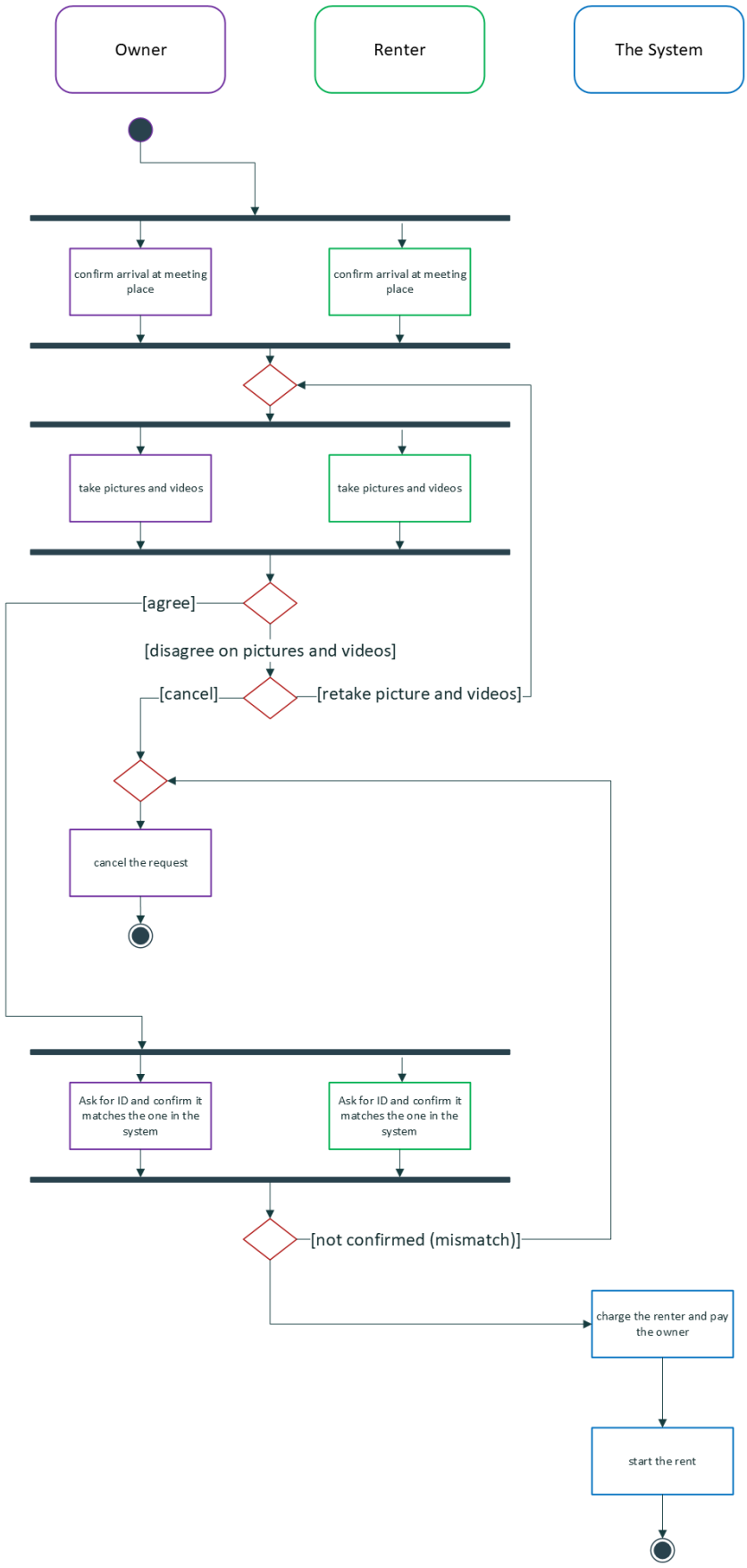

Figure 3 - Activity diagram of delivering a tool

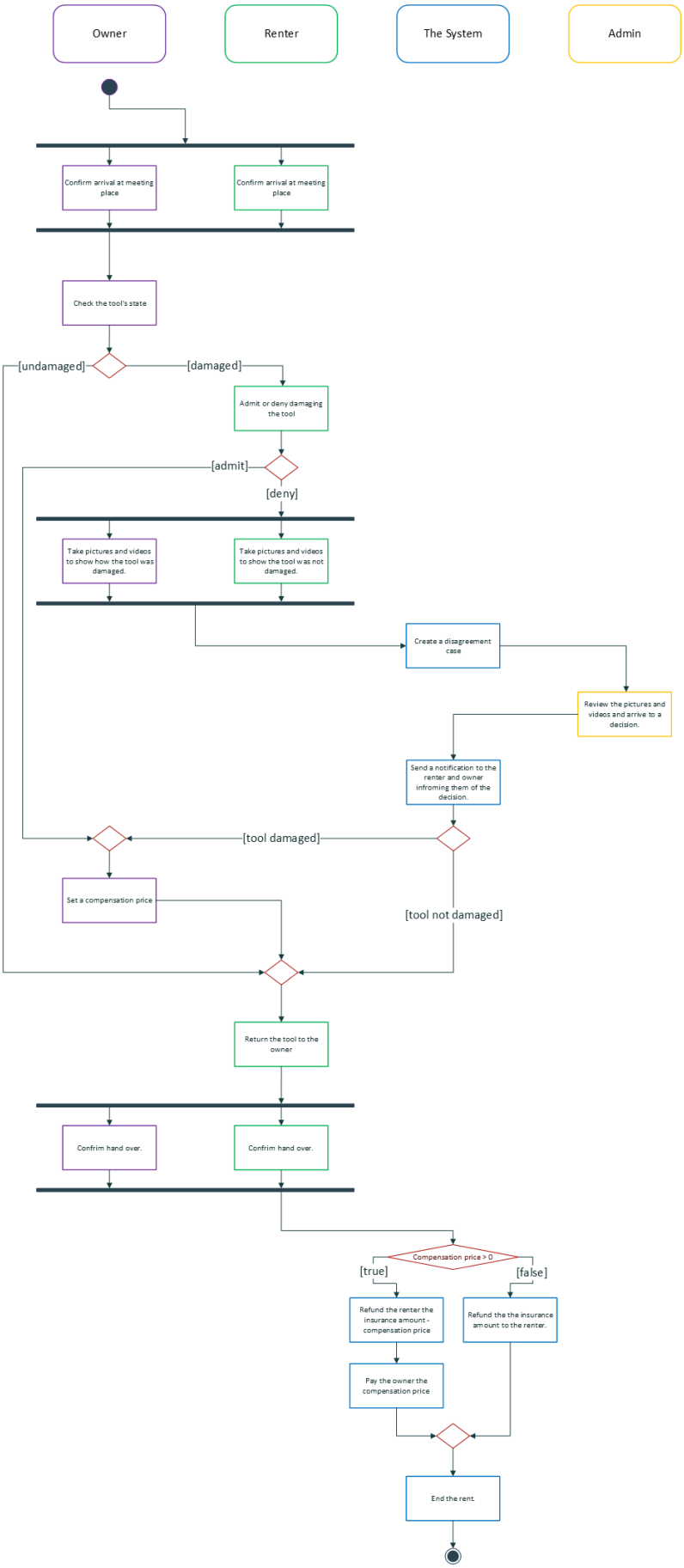

Figure 4 - Activity diagram of returning a tool

3.1.5 Class diagram

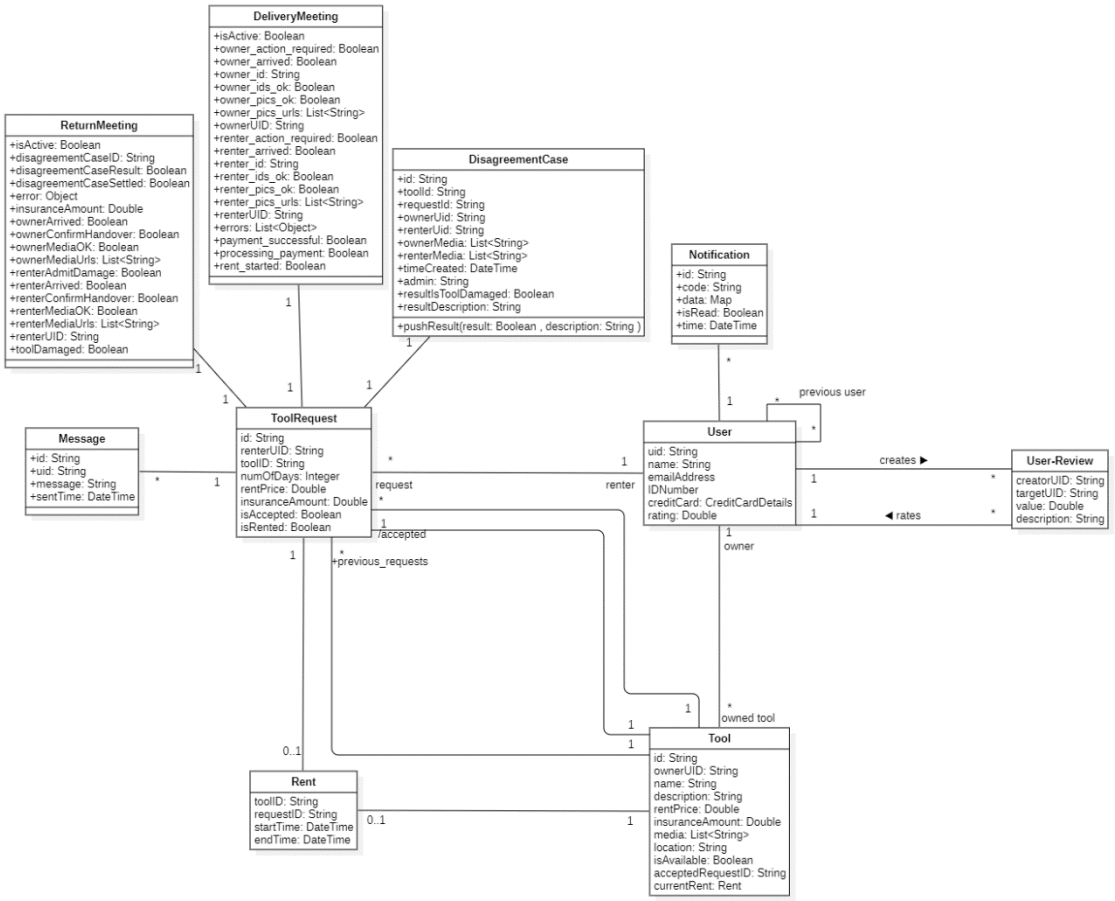

Figure 5 - The Conceptual model

3.2 Functional Requirements

3.2.1. Functional requirement 1

- **ID:** FR1
- **BUC:** UC1, UC2, UC5, UC6, UC8, UC9, UC10
- **Description:** The system must accept a range of inputs from the user
- **Fit Criterion:** A range of inputs will be accepted
- **Rationale:** To be able to read inputs from the user
- **Dependencies:** None
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.2. Functional requirement 2

- **ID:** FR2
- **BUC:** UC8
- **Description:** The system must allow the user to create an account to log into.
- **Fit Criterion:** The Sign-up screen will be displayed
- **Rationale:** To be able to identify the user and allow the user to access the system functionalities
- **Dependencies:** FR1
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.3. Functional requirement 3

- **ID:** FR3
- **BUC:** UC8-B
- **Description:** The system must allow the user to log-in/sign-up using a Google, Facebook, Microsoft, or Apple account.
- **Fit Criterion:** "Log-in with" buttons will be displayed in the log-in and sign-up screens
- **Rationale:** To add an easier and quicker method of singing in
- **Dependencies:** FR1, FR2
- **Priority:** Low
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.4. Functional requirement 4

- **ID:** FR4
- **BUC:** UC8

- **Description:** The system must allow the user to enter his/her ID number.
- **Fit Criterion:** a field to enter ID number will be displayed
- **Rationale:** To ensure the user's identity to prevent scams
- **Dependencies:** FR1, FR2
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.5. Functional requirement 5

- **ID:** FR5
- **BUC:** UC9
- **Description:** The system must allow the user to log into his/her account
- **Fit Criterion:** The log-in screen will be displayed
- **Rationale:** To be able to identify the user and allow the user to access the system functionalities
- **Dependencies:** FR1, FR2
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.6. Functional requirement 6

- **ID:** FR6
- **BUC:** UC9
- **Description:** The system must be check if the log-in details are correct
- **Fit Criterion:** A wrong log-in details will be rejected
- **Rationale:** To prevent unauthorized access to a user's account
- **Dependencies:** FR2
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.7. Functional requirement 7

- **ID:** FR7
- **BUC:** UC11
- **Description:** The system must allow the admins to ban a user
- **Fit Criterion:** The banned user's email address and ID number will be added to the banned list
- **Rationale:** To prevent the banned user from ever accessing the system's functionalities
- **Dependencies:** FR1, FR2
- **Priority:** Medium

- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: Admin requirement

3.2.8. Functional requirement 8

- **ID:** FR8
- **BUC:** UC8, UC9
- **Description:** The system must be able to check if the user is banned
- **Fit Criterion:** A user's email or ID that's on the banned list will be detected and his/her request will be denied
- **Rationale:** To prevent a banned user from accessing the system
- **Dependencies:** FR7, FR2
- **Priority:** Medium
- **Support materials:** None
- **History:** Created 4/3/2021
  Edited 9/3/2021
- **Type**: User requirement

3.2.9. Functional requirement 9

- **ID:** FR9
- **BUC:** UC1, UC4, UC5, UC8
- **Description:** The system must allow the user to enter a valid credit card detail.
- **Fit Criterion:** fields to enter card details will be displayed
- **Rationale:** To be able to send and receive payments
- **Dependencies:** FR1, FR2
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
  Edited 9/3/2021
- **Type**: User requirement

3.2.10. Functional requirement 10

- **ID:** FR10
- **BUC:** UC1, UC4, UC5, UC8
- **Description:** The system must be able to determine if a card is valid or not
- **Fit Criterion:** an invalid card will be denied.
- **Rationale:** To prevent fake cards
- **Dependencies:** FR9
- **Priority:** High
- **Support materials:** None
- **History:** Created 9/3/2021
- **Type**: System requirement

3.2.11. Functional requirement 11

- **ID:** FR11
- **BUC:** UC1
- **Description:** The system must allow the user (owner) to create a new post
- **Fit Criterion:** a screen to create a new post will be displayed
- **Rationale:** To rent their tools, and for renters to send them tool-requests
- **Dependencies:** FR1, FR2, FR12
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: Owner requirement

3.2.12. Functional requirement 12

- **ID:** FR12
- **BUC:** UC1
- **Description:** The system must be able to determine if a user is authorized to create a new post.
- **Fit Criterion:** The user will get a prompt telling them they can't post if they are not logged in or didn't provide an ID number and card details.
- **Rationale:** To prevent unauthorized accounts from creating posts and filling the database
- **Dependencies:** FR2, FR4
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: Owner requirement

3.2.13. Functional requirement 13

- **ID:** FR13
- **BUC:** UC2
- **Description:** The system must allow the owner to edit and delete their posts
- **Fit Criterion:** a screen to edit the post will be displayed with a button to delete the post
- **Rationale:** To prevent unauthorized users from posting
- **Dependencies:** FR1, FR2, FR11
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
  Edited 9/3/2021
- **Type**: Owner requirement

3.2.14. Functional requirement 14

- **ID:** FR14
- **BUC:** UC2
- **Description:** The system must be able to determine if a user is authorized to edit or remove a post
- **Fit Criterion:** The user's request to edit or delete will be denied if (s)he is not authorized to
- **Rationale:** To prevent the posts from being edited by an unauthorized user
- **Dependencies:** FR2, FR11, FR13
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: Owner requirement

3.2.15. Functional requirement 15

- **ID:** FR15
- **BUC:** UC3,
- **Description:** The system must be able to send notifications to a user.
- **Fit Criterion:** The user will receive a notification
- **Rationale:** To alert the user of any important information
- **Dependencies:** -
- **Priority:** Medium
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.16. Functional requirement 16

- **ID:** FR16
- **BUC:** UC3
- **Description:** The system must be able to determine if a user is authorized to view, accept, and deny tool-requests to a certain post.
- **Fit Criterion:** an unauthorized user will not be able to view, accept, or deny a tool-request
- **Rationale:** To either rent the tool to the renter or refuse his request and remove it from the tool-requests list
- **Dependencies:** FR2, FR11, FR21
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: Owner requirement

3.2.17. Functional requirement 17

- **ID:** FR17
- **BUC:** UC3

- **Description:** The system must allow the owner to view tool-requests sent to their posts
- **Fit Criterion:** a screen with all the requests will be displayed
- **Rationale:** For the owner to accept it and rent the tool to the renter
- **Dependencies:** FR2, FR11, FR16, FR21
- **Priority:** High
- **Support materials:** None
- **History:** Created 4/3/2021
  Edited 9/3/2021
- **Type**: Owner requirement

3.2.18. Functional requirement 18

- **ID:** FR18
- **BUC:** UC3
- **Description:** The system must allow the owner to accept or reject a tool-request
- **Fit Criterion:** the request will be accepted. Or the request will be rejected and the tool-request will be deleted.
- **Rationale:** To start renting the tool to the renter. Or reject a renter's offer
- **Dependencies:** FR2, FR11, FR16, FR17, FR21
- **Priority:** High
- **Support materials:** Figure 2
- **History:** Created 4/3/2021
  Edited 9/3/2021
- **Type**: Owner requirement

3.2.19. Functional requirement 19

- **ID:** FR19
- **BUC:** UC3
- **Description:** The system must be able to open a chat room between the owner and renter
- **Fit Criterion:** a chat room will be open where the owner and renter can send and receive messages from each other
- **Rationale:** For the owner and renter to communicate
- **Dependencies:** FR1, FR2
- **Priority:** Low
- **Support materials:** None
- **History:** Created 4/3/2021
- **Type**: User requirement

3.2.20. Functional requirement 20

- **ID:** FR20
- **BUC:** UC5, UC6

- **Description:** The system must be able to determine if a user is authorized to send, edit, or remove a tool-request to a post.
- **Fit Criterion:** The user's request to add, edit or delete a tool-request will be denied if (s)he is not authorized to
- **Rationale:** To prevent unauthorized access to a tool-request
- **Dependencies:** FR2, FR4
- **Priority:** High
- **Support materials:** None
- **History:** Created 5/3/2021
- **Type**: Renter requirement

3.2.21. Functional requirement 21

- **ID:** FR21
- **BUC:** UC5
- **Description:** The system must allow the renter to send a tool-request to a post.
- **Fit Criterion:** a tool-request screen will be displayed
- **Rationale:** For the owner to accept it and rent the tool to the renter
- **Dependencies:** FR2, FR11, FR20
- **Priority:** High
- **Support materials:** None
- **History:** Created 5/3/2021
  Edited 9/3/2021
- **Type**: Renter requirement

3.2.22. Functional requirement 22

- **ID:** FR22
- **BUC:** UC6
- **Description:** The system must allow the renter to edit or remove his/her tool-request
- **Fit Criterion:** a screen to edit the request will be displayed with a button to delete the tool-request
- **Rationale:** To allow the renter to change his tool-request details or delete the request entirely
- **Dependencies:** FR2, FR20, FR21
- **Priority:** High
- **Support materials:** None
- **History:** Created 5/3/2021
  Edited 9/3/2021
- **Type**: Renter requirement

3.2.23. Functional requirement 23

- **ID:** FR23
- **BUC:** UC10

- **Description:** The system must be able to determine if two users had a previous rental-transaction.
- **Fit Criterion:** the system will confirm if two users had a previous transaction or not.
- **Rationale:** To allow them to rate each other
- **Dependencies:** FR2
- **Priority:** Low
- **Support materials:** None
- **History:** Created 5/3/2021
- **Type**: User requirement

3.2.24. Functional requirement 24

- **ID:** FR24
- **BUC:** UC10A
- **Description:** The system must allow the user to add/edit a review on another user if they had a previous transaction
- **Fit Criterion:** A form to submit the review details will be displayed
- **Rationale:** To rate his/her experience with the other user
- **Dependencies:** FR1, FR2, FR23
- **Priority:** Low
- **Support materials:** None
- **History:** Created 5/3/2021
- **Type**: User requirement

3.2.25. Functional requirement 25

- **ID:** FR25
- **BUC:** UC10B
- **Description:** The system must be able to determine if a user is authorized to delete a review
- **Fit Criterion:** the delete request will be denied if the user isn't authorized
- **Rationale:** To prevent an unauthorized user from deleting a review
- **Dependencies:** FR2, FR24
- **Priority:** Low
- **Support materials:** None
- **History:** Created 5/3/2021
- **Type**: User requirement

3.2.26. Functional requirement 26

- **ID:** FR26
- **BUC:** UC10B
- **Description:** The system must allow the user to delete a review.
- **Fit Criterion:** an option (button) to delete the review will be displayed
- **Rationale:** To remove the review from the user's profile
- **Dependencies:** FR2, FR24, FR25

- **Priority:** Low
- **Support materials:** None
- **History:** Created 5/3/2021
- **Type**: User requirement

3.2.27. Functional requirement 27

- **ID:** FR27
- **BUC:** UC4
- **Description:** The system must allow the user to take pictures and videos.
- **Fit Criterion:** The camera screen will be displayed
- **Rationale:** To take pictures and videos of a tool
- **Dependencies:** FR1, FR2
- **Priority:** High
- **Support materials:** Figure 3
- **History:** Created 9/3/2021
- **Type**: User requirement

3.2.28. Functional requirement 28

- **ID:** FR28
- **BUC:** UC4
- **Description:** The system must allow the user to upload pictures and videos.
- **Fit Criterion:** The pictures and videos will be uploaded to the server.
- **Rationale:** to upload pictures and videos of a tool
- **Dependencies:** FR27
- **Priority:** High
- **Support materials:** Figure 3
- **History:** Created 9/3/2021
- **Type**: User requirement

3.2.29. Functional requirement 29

- **ID:** FR29
- **BUC:** UC4
- **Description:** The system must be able to send and receive payment
- **Fit Criterion:** The money will be added/deducted to/from the system's account balance.
- and vice versa for the user's account.
- **Rationale:** To send money between users and from the system to the users
- **Dependencies:** FR4
- **Priority:** High
- **Support materials:** None
- **History:** Created 9/3/2021
- **Type**: System requirement

3.2.30. Functional requirement 30

- **ID:** FR30
- **BUC:** UC1,2,3,5, and 6
- **Description:** The system must be able to display a prompt to the user
- **Fit Criterion:** The prompt will be displayed
- **Rationale:** To inform the user of specific information
- **Dependencies:** -
- **Priority:** High
- **Support materials:** None
- **History:** Created 9/3/2021
- **Type**: User requirement

3.2.31. Functional requirement 31

- **ID:** FR31
- **BUC:** UC7
- **Description:** The system must be able to create a disagreement case. It's created any time the user and owner have a disagreement. And it needs to be reviewed by the admin.
- **Fit Criterion:** The case will be created in the database where it will be available to the admin
- **Rationale:** For the admin to review the problem and arrive to a decision
- **Dependencies:** FR2
- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
- **Type**: System requirement

3.2.32. Functional requirement 32

- **ID:** FR32
- **BUC:** UC7
- **Description:** The system must allow the admin to review a disagreement case and arrive to a decision.
- **Fit Criterion:** A screen for the disagreement case will be displayed and it contains a from to fill the decision
- **Rationale:** So, the admin can settle and fix the problem
- **Dependencies:** FR2
- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
- **Type**: Admin requirement

3.2.33. Functional requirement 33

- **ID:** FR33
- **BUC:** UC7

- **Description:** The system must allow the owner to claim the tool is damaged.
- **Fit Criterion:** An option or a button will be available to start the claim.
- **Rationale:** To prevent the renter from damaging the tool without paying for it
- **Dependencies:** FR2
- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
  Edited 9/3/2021
- **Type**: Owner requirement

3.2.34. Functional requirement 34

- **ID:** FR34
- **BUC:** UC7
- **Description:** The system must be able to calculate the total money needed from/to a user. Including the remaining of the insurance money.
- **Fit Criterion:** The total will be calculated and displayed
- **Rationale:** To send the appropriate amount of money to each the owner and renter
- **Dependencies:** -
- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
- **Type**: System requirement

3.2.35. Functional requirement 35

- **ID:** FR35
- **BUC:** UC4, UC7
- **Description:** The system must allow the user to upload documents.
- **Fit Criterion:** A file upload screen will be displayed
- **Rationale:** To upload any official documents
- **Dependencies:** FR1, FR2
- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
- **Type**: User requirement

3.2.36. Functional requirement 36

- **ID:** FR36
- **BUC:** UC7
- **Description:** The system must allow the renter and owner to agree on a compensation price
- **Fit Criterion:** The agreement price screen will be displayed
- **Rationale:** So, the owner and renter can agree on a middle ground
- **Dependencies:** FR1, FR2

- **Priority:** High
- **Support materials:** Figure 4
- **History:** Created 9/3/2021
- **Type**: User requirement

3.2.37. Functional requirement 37

- **ID:** FR37
- **BUC:** None
- **Description:** The system must allow the user to search for a tool post.
- **Fit Criterion:** A search bar will be displayed.
- When the user searches, a list of posts that match the search query will be displayed
- **Rationale:** To find a post of a specific tool so the renter can send a tool-request to.
- **Dependencies:** FR1
- **Priority:** High
- **Support materials:** None
- **History:** Created 9/3/2021
- **Type**: User requirement

3.3 Non-Functional Requirements

3.3.1 Look and feel requirements

3.3.1.1 Material design

- **ID:** LFR1
- **Description:** The system must follow the material design guidelines.
- **History:** Created 4/4/2021
- **Type:** Look and feel

3.3.1.2 Colors

- **ID:** LFR2
- **Description:** The system design colors must rely mainly on the two primary colors of the brand while following LFR1.
- **History:** Created 4/4/2021
- **Type:** Look and feel requirement

3.3.2 Usability and humanity requirements

3.3.2.1 Ease of use

- **ID:** UR1
- **Description:** The system must be easy to use and learn by different users.
- **History:** Created 4/4/2021
- **Type:** Usability and humanity requirement

3.3.2.2 Localization

- **ID:** UR2
- **Description:** The system must be available in Arabic and English
- **History:** Created 4/4/2021
- **Type:** Usability and humanity requirement

3.3.3 Performance requirements

3.3.3.1 Load handling

- **ID:** PR1
- **Description:** The system must handle up to 500 users simultaneously
- **History:** Created 4/4/2021
- **Type:** Performance requirements

3.3.3.2 Inputs respond time

- **ID:** PR2
- **Description:** The system must respond to most inputs within 2 seconds
- **History:** Created 4/4/2021
- **Type:** Performance requirements

3.3.3.3 Complex respond time

- **ID:** PR3
- **Description:** The system must respond to complex requests within 20 seconds
- **History:** Created 4/4/2021

- **Type:** Performance requirements

3.3.3.4 Availability

- **ID:** PR4
- **Description:** The system must be available at all times.
- **History:** Created 4/4/2021
- **Type:** Performance requirements

3.3.4 Maintainability and support requirements

3.3.4.1 Future Expansions

- **ID:** MR1
- **Description:** The system must be able to cope with expansions to global availability.
- **History:** Created 9/4/2021
- **Type:** Maintainability and support requirement

3.3.4.2 Extra localization plans

- **ID:** MR2
- **Description:** The system must be able to add support for different languages.
- **History:** Created 4/4/2021
- **Type:** Maintainability and support requirement

3.3.5 Legal requirements

3.3.5.1 Law-abiding system

- **ID:** LR1
- **Description:** The system must operate in accordance with Saudi Arabia’s laws.
- **History:** Created 4/4/2021
- **Type:** Legal requirement

3.4 Analysis

3.4.1 Constraints on objects attributes

3.4.1.1 Constraints on User objects:

- **Context** User **inv:**
  self.uid == self.ownedTool.ownerUID

- **Context** User **inv:**
  self.uid == self.request.renterID

- **Context** User **inv:**
  self.uid <> Self.request.tool.ownerUID

- **Context** User **inv:**
  self.rating >= 0 **AND** self.rating <= 5

3.4.1.2 Constraints on ToolsRequest objects:

- **Context** ToolsRequest **inv:**
  Self.isAccepted == true **implies** self.id == self.tool.acceptedRequestID

- **Context** ToolsRequest **inv:**
  self.renterUID <> self.tool.ownerUID

- **Context** ToolsRequest **inv:**
  self.numOfDays > 0

3.4.1.3 Constraints on Rent objects:

- **Context** Rent **inv**:
- self.toolID == self.request.toolID

- **Context** Rent **inv**:
- self.tool == self.request.tool

- **Context** Rent **inv**:
- self.endTime **after** self.startTime

3.4.1.4 Constraints on UserReview objects:

- **Context** UserReview **inv**:
- self.creatorUID <> self.targetUID

- **Context** UserReview **inv**:
- self.value >= 0 **AND** self.value <= 5

3.4.1.5 Constraints on Tool objects:

- **Context** Tool **inv**:
  self.requests **include** self.acceptedRequest

- **Context** Tool **inv:**
  self.owner.tools **include** self

3.4.2 System Operations

Below are the main system operations associated with the system classes

3.4.2.1 User.createPost()

Context User:: createPost(name: String, description: String, rentPrice: Double, insuranceAmount: Double, media: List<String>, location: String)

Pre:

- name.length > 0
- rentPrice > 0
- insuranceAmount > 0
- location.length > 0

Post:

- a new Tool object will be created.
- the user will be linked with the Tool object.
- if the operation succeeded the Tool object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.2 Tool.edit()

Context Tool::edit(name: String, description: String, rentPrice: Double, insuranceAmount: Double, media: List<String>, location: String)

Pre:

- name.length > 0
- rentPrice > 0
- insuranceAmount > 0
- location.length > 0

Post:

- the Tool object will be edited.
- if the operation succeeded the Tool object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.3 Tool.acceptRequest()

Context Tool::acceptRequest(toolRequestID)

Pre:

- Tool.requests must include toolRequestID.
- Tool. acceptedRequestID == null

Post:

- Tool.acceptedRequestID will become toolRequest.id
- A chat room will be created

- The Tool owner will be linked with the chat room
- The ToolRequest sender (renter) will be linked with the chat room
- The ToolRequest sender (renter) will be notified
- if an error occurred a string indicating the error will be returned.

3.4.2.4 Tool.startRent()

Context Tool :: startRent()

Pre:

- The tool must have an accepted tool-request.

Post:

- A new Rent object, rent, will be created as follows:
  - rent.toolID will become tool.ID
  - rent.requestID will become tool.acceptedRequestID
  - rent.startTime will become DateTime.now()
- The Rent object will be linked with the Tool
- tool.currentRent will become *rent*
- tool.acceptedToolRequest.isRented will become true
- The Rent object will be linked with the ToolRequest
- if the operation succeeded the Rent object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.5 Tool.addRequest()

Context Tool:: addRequest(renterUID: String, toolID: String, numOfDays: Integer, rentPrice: Double, insuranceAmount: Double)

Pre:

- the renter must not have a previous request on the tool.
- renterUID is a valid user ID
- toolID is a valid tool ID
- toolID doesn't reference a tool where tool.ownerID == renterID (i.e., a user can't request his/her own tool)
- numOfDays > 0
- rentPrice > 0
- insuranceAmount > 0

Post:

- a new ToolRequest object will be created.
- The ToolRequest and tool will be linked
- The ToolRequest object will be linked with the renter
- The ToolRequest object will be linked with the Tool object
- if the operation succeeded the ToolRequest object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.6 ToolRequest.edit()

Context ToolRequest:: edit(numOfDays: Integer, rentPrice: Double, insuranceAmount: Double)

Pre:

- numOfDays > 0
- rentPrice > 0
- insuranceAmount > 0

Post:

- the ToolRequest object will be edited.
- if the operation succeeded the ToolRequest object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.7 Tool.endRent()

Context Tool:: endRent()

Pre:

- currentRent <> null (i.e., there must be a rent in progress)

Post:

- tool.currentRent will be deleted and become null.
- tool.acceptedToolRequest will be deleted.
- tool.acceptedRequestID will become null
- if an error occurred a string indicating the error will be returned.

3.4.2.8 User.addReview()

Context User:: addReview(value: Double, description: String)

Pre:

- user.hadAPreviousRentWith() == true
- value >= 0 AND value <= 5 (i.e., value must be between 0 and 5)

Post:

- a new Review object, review, will be created
- review will be linked to user
- review will be linked to target
- if the operation succeeded the Review object's document-reference will be returned.
- otherwise (if an error occurred) a string indicating the error will be returned.

3.4.2.9 User.ban()

Context User:: ban()

Pre:

- initiator.isAdmin == true (i.e., the initiator must be an administrator)

Post:

- user.IDNumber will be added to the banned list
- user.emailAddress will be added to the banned list
- any tool-requests from the user will be delete

3.4.3 Analysis class diagram

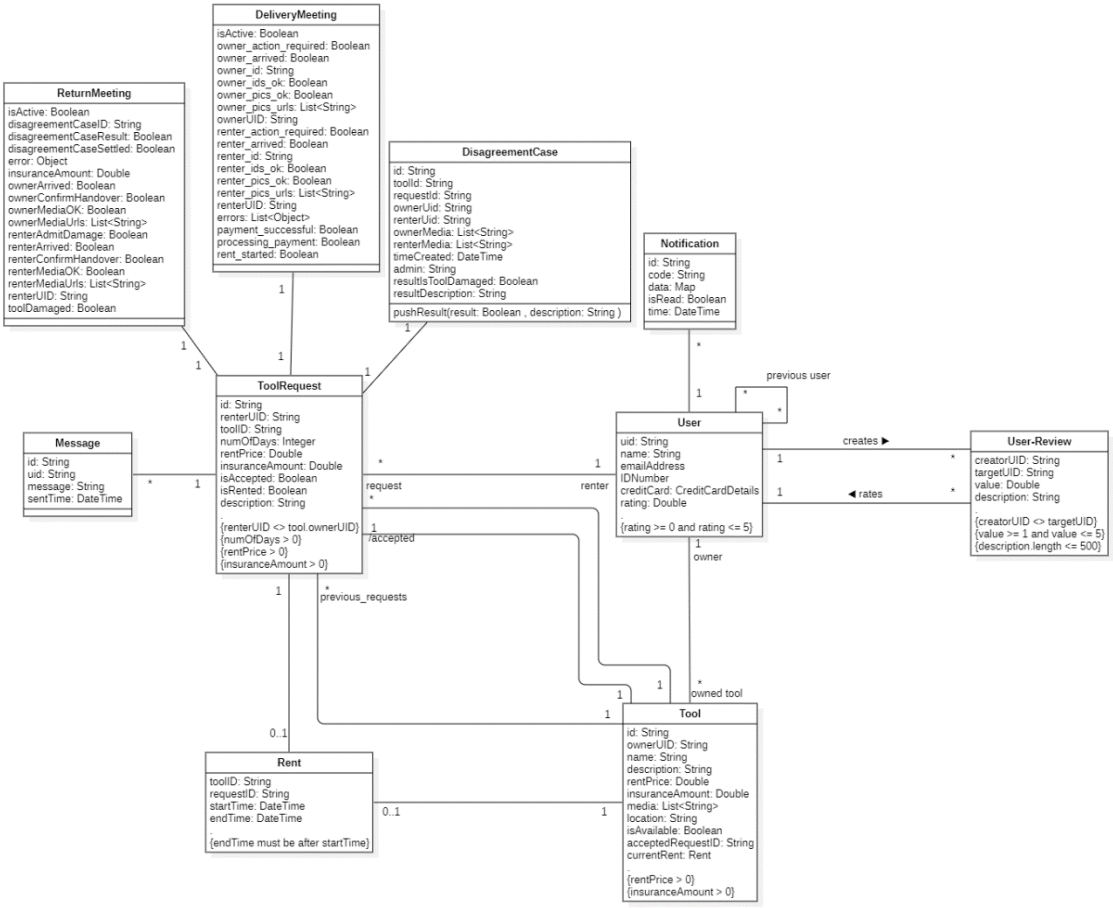

Figure 6 - Analysis class diagram

3.5 Glossary

Term	*Category*	*Definition*
Owner	Role	The user that is offering his tool for rent.
Renter	Role	The user that is requesting to rent an owner's tool.
Admin	Role	The system administrator. (S)He has access to all functions and operations.
Post	Technical	An online page for a *Tool*, where users can find it and send tool-requests on it.
User	Concept	A person that is using the system, includes *Owner, Renter,* and *Admin.*
Tool	Concept	A tool that a *User* has offered for rent.
ToolRequest	Concept	A request from the renter to rent the owner's tool. Where the owner can reject or accept it.
Rent	Concept	A renting process that is created after delivering a *Tool* to the *Renter*.
UserReview	Concept	A review/rating from a *User* to another *User.*
Ban a *User*	Operation	Prevent a *User* from accessing the system functionalities and operations
Document-reference	Technical	A *DocumentReference* refers to a document location in a database and can be used to write, read, or listen to the location.

3.5.2 Glossary of class attributes

Term	*Category*	*Definition*
uid	Attribute of *User*	User ID: a unique identifier for a *User.*
name	Attribute of *User*	The name of a *User*.
emailAddress	Attribute of *User*	The email address of a *User.*
IDNumber	Attribute of *User*	The ID number of a *User.*
creditCard	Attribute of *User*	Credit card details of a *User.*
rating	Attribute of *User*	A value that rates a *User* out of 5, it's the average of all *User.reviews*
reviews	Attribute of *User*	A collection of *UserReview* objects that rates a *User.*
tools	Attribute of *User*	A collection of *Tool* objects that belong to the *User*.

requests	Attribute of *User*	A collection of *ToolRequest*.
id	Attribute of *Tool*	A unique identifier for a *Tool*.
ownerUID	Attribute of *Tool*	A unique identifier for the *User* that owns a *Tool* (i.e., the *Owner.uid*).
name	Attribute of *Tool*	The name of a *Tool*.
description	Attribute of *Tool*	A description of a *Tool.*
rentPrice	Attribute of *Tool*	A *Tool* rent price for one day.
insuranceAmount	Attribute of *Tool*	An insurance amount of a *Tool* that is held on the *Renter*'s credit card until the *Tool* is returned in a good condition.
media	Attribute of *Tool*	A List of URLs of pictures and/or videos of a *Tool.*
location	Attribute of *Tool*	The country and city of a *Tool.*
isAvailable	Attribute of *Tool*	An indicator if a *Tool* is available to send requests to.
acceptedRequestID	Attribute of *Tool*	A unique identifier for the accepted *ToolRequest* on a *Tool.*
currentRent	Attribute of *Tool*	A *Rent* object that is created after delivering the *Tool* to the *Renter*.
requests	Attribute of *Tool*	A collection of *ToolRequest*s that are sent to a *Tool.*
id	Attribute of *ToolRequest*	A unique identifier for a *ToolRequest*.
renterUID	Attribute of *ToolRequest*	A unique identifier for the *User* that sent a *ToolRequest* (i.e., the *Renter.uid*).
toolID	Attribute of *ToolRequest*	A unique identifier for the *Tool* a *ToolRequest* is sent to.
numOfDays	Attribute of *ToolRequest*	The number of days a *Renter* want to rent the tool for.
rentPrice	Attribute of *ToolRequest*	A *Tool* rent price for one day.
insuranceAmount	Attribute of *ToolRequest*	An insurance amount of a *Tool* that is held on the *Renter*'s credit card until the *Tool* is returned in a good condition.
isAccepted	Attribute of *ToolRequest*	An indicator if a *ToolRequset* is accepted or not.
isRented	Attribute of *ToolRequest*	An indicator if a *ToolRequset* is accepted and the *Tool* is delivered, and a *Rent* has started.
toolID	Attribute of *Rent*	A unique identifier for a *Tool* linked to a *Rent*.

requestID	Attribute of *Rent*	A unique identifier for a *ToolRequest* linked to a *Rent.*
startTime	Attribute of *Rent*	the start date and time of a *Rent.* (i.e., the date and time of delivering the *Tool*).
endTime	Attribute of *Rent*	the end date and time of a *Rent.* (i.e., the date and time of returning the *Tool*).
creatorUID	Attribute of *UserReviews*	a unique identifier for a *User* that is creating a *UserReview*.
targetUID	Attribute of *UserReviews*	a unique identifier for a *User* that a *UserReview* is rating/targeting.
value	Attribute of *UserReviews*	A value that rates a *User* out of 5, it's the average of all *User.reviews*.
description	Attribute of *UserReviews*	A description of a *UserReview*.

Chapter 4 – Design, Implementation, and testing

4.1 Development model

I chose the iterative and incremental development model to develop this system. The reason was my lack of experience developing e-commerce and rental systems. And most importantly because I didn't completely know how I was going to build the system with the features I had in mind such as delivery and return meetings and ensuring their flow in the correct order and disagreement cases and how are they going to be built and settled. The iterative and incremental model allowed me to incrementally grow the system and learn how to build it with each iteration of building until it was complete.

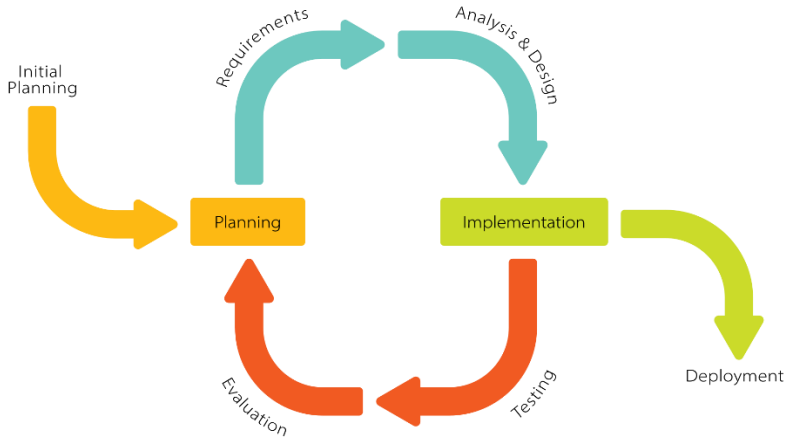

Figure 7 Development life cycle. By (Krupadeluxe, 2021)

4.2 Identity

The system is called "Rentool", and the two main colors are blue (#2196F3 ■) and white. The logo is designed with Figma using free-to-use vectors from Material Design by Google and the typeface font family is NATS.

Figure 8 Rentool logo

4.3 System Infrastructure

4.3.1 Frontend

For the front-end, the app was developed and built using the Flutter framework by Google. Flutter makes it possible to build fast and beautiful apps for Android, iOS, the web, and other platforms with a single codebase. The programming language used is Dart which also has its own package manager, *Pub*.

I chose Flutter because it's great for this system as it can be compiled to Android, iOS, and a web app. Also, I'm very familiar with it and used it to build multiple mobile and web apps. It has a lot of support and resources online. And there are official packages for Firebase SDKs and APIs which I'm using for the backend of the system.

For UI design, I used Figma. Figma is a popular free-to-use vector graphics editor and prototyping tool that and I used to design the app screens, buttons, widgets, and logo.

4.3.2 Backend

The system is deployed in the cloud using *Google Firebase* with the project id "rentool-5a78c". *Firebase* is an app development platform that offers a lot of features to build and deploy mobile and web apps.

I chose Firebase because I'm very familiar with it and developed multiple apps using it. It also allows me to not worry about implementing some of the services it provides (e.g., database, authentication) from scratch and stressing about making them work with each other. Instead, I can focus on building the business side of the system.

4.2.2.1 Authentication

For the authentication system, the system uses *Firebase Authentication* to manage and handle users' authentication. It allows users to create accounts using several methods, like the email and password, or with a phone number. It also allows signing users in using other providers such as Google, Facebook, Microsoft, Twitter, and much more. It can handle email verifications, password resets, and more with very little effort from the developer. This provides a strong and reliable authentication system powered and backed up by Google.

For this system, the sign-in methods used are email and password, Google, and Facebook. I couldn't add Apple and Microsoft as was intended in FR3 because Apple requires an annual $99 for a developer account which I couldn't pay. And I had technical issues with Microsoft due to the lack of documentation and libraries for Flutter.
Once a user creates an account the account will be assigned a unique UID (short for User ID) which will be used as a reference to this user's account.

4.2.2.2 The Database

Firestore is used for the system's database. Firestore is a NoSQL database that stores the data in documents where the data is stored as a field-value map (like JSON) and each document is stored in a collection which is like a container for documents.

Firestore starts with a list of collections that each contains a list of documents. Each document has an ID that it can be accessed by. And each document has a list of fields that can be one of several types (e.g., string, bool, number), and has a list of collections (subcollections). The two main collections in this system are 'Users' and 'Tools'.

'Users' contains the users' documents with their UIDs (User ID) as the documents' IDs, such as you can access the document of a user with the UID 'user123' for example, by querying '`Users/user123`'. A user document has the user's profile info such as the name, profile pic, rating …etc.

The user document has multiple subcollections, including 'private' which stores data that can only be accessed by the user. It has only three documents, 'ID', 'card', and 'checklist'. 'ID' stores the national ID number of the user. 'card' stores the sharable card details (e.g., last 4 numbers, bin, type) of the user's credit/debit card. The 'checklist' document stores a checklist of whether the user has an ID number, has a credit/debit card and whether the card supports payouts (i.e., sending money directly **to** the card). 'checklist' is automatically updated and is used to check all of these things without querying multiple documents. The user's document also has other

subcollections like ‘reviews’, ‘notifications’, ‘devices’, and others which will all be explained later.

The ‘Tools’ collection contains the tools’ documents, and every tool document is linked to a user using the `ownerUID` field. The tool’s document contains all its public data (e.g., name, description, rent price), and has the subcollections `requests`, `previous_requests`, `deliver_meetings`, and `return_meetings`. `requests` store the tool requests sent to the tool. `previous_requests` store the requests of previous rents on the tool. `deliver_meetings` and `return_meetings` are for delivery meetings and return meetings documents respectively.

The other collections are:

- ‘rents’ which stores current and previous rent objects (created when the rent starts, and the renter takes the tool).
- ‘disagreementsCases’ stores documents of the disagreement cases between users for admins to then review them and be the judge in them.
- ‘idsList’ stores all the users’ ID numbers. This is to prevent users from making multiple accounts with the same id number, where the system checks if the ID number entered by the user already exists in this collection or not, and if it doesn’t, it gets added.
- ‘bannedList’ contains all the banned national ID numbers. Once a user is banned, if he/she has a national ID number set, the number will be added to this collection with the reason for the ban and the admin who banned him/her. Then, if the user created another account and tried the same national ID number it'll be rejected.
- ‘bannedUsers’ is similar to ‘bannedList’, but it stores the banned user’s UID as the document id instead. This collection is useful for banned users who didn’t set their ID numbers. This is primarily used to store reasons behind bans.
- ‘admins’ stores the admins' UIDs. Each document’s ID is an admin UID. When a document is added the user will be assigned as an admin. And the opposite if a document was removed.
- ‘payment_references’ store reference documents to describe the purpose of a payment request. The IDs of the documents in this collection is used as reference code to the payment requests sent to Checkout API.

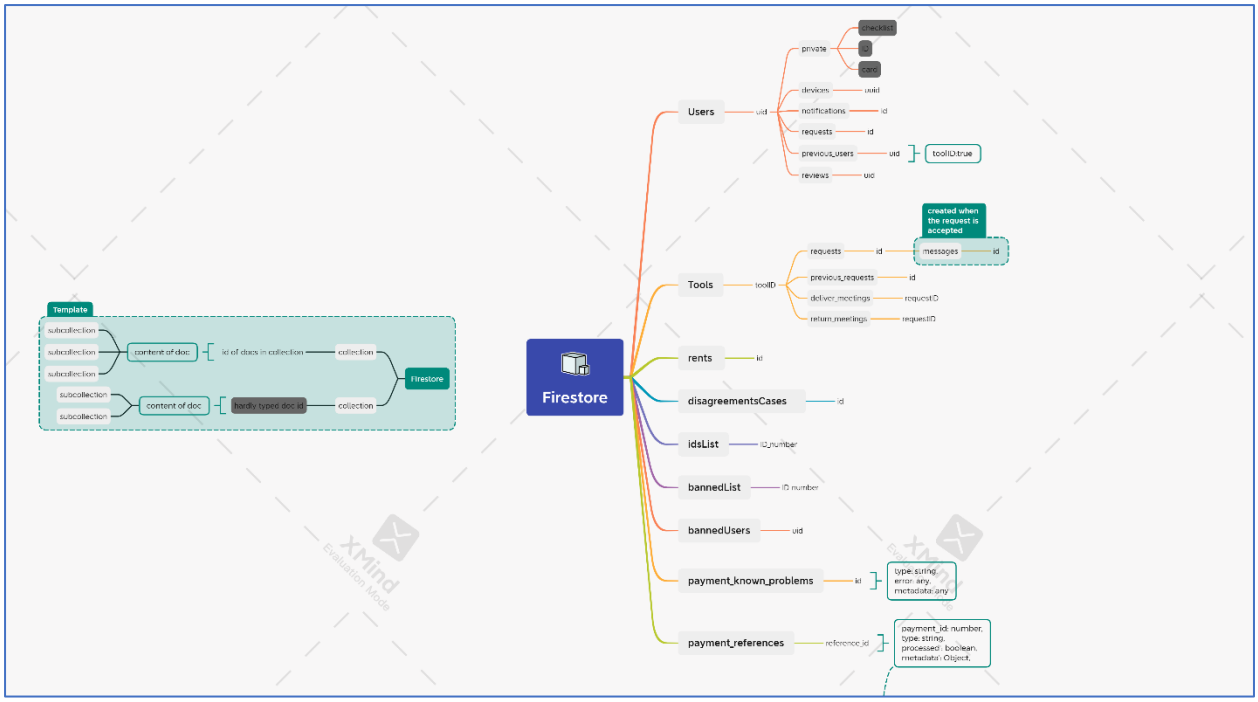

Figure 9 A simplified map of the database

Reading and writing to a document is done in the code using the Firestore SDK, it's done by getting a document reference and then calling the wanted method (e.g., `.get()`, `.update()`). The SDK will automatically add the user authentication token if he/she is signed in. and then Firestore will determine if the user has access to read or write the document. For example, a call to get the document of the tool with id "tool_123" looks like this:

```
FirebaseFirestore.instance.doc('Tools/tool_123').get();
```

So, access control is determined by the database's *Security Rules.* Security rules are a set of rules that control access to documents and collections in Firestore. The way these rules work is by a `match` statement specifying a document path and an `allow` expression detailing when reading or writing the specified data is allowed.

Security rules can also be used for data validation such as making sure a field is set, or the value of a certain field is of the correct type. This is very necessary to make sure all the objects are of the correct type and prevent type cast errors.

Example of simplified security rules for the 'Users' collection.

```
// This match statemtns matches any document in the 'Users' collection.
// Curly brakets are wildcards. Which makes this statement matches any document in the 'Users'
// collection (e.g., 'Users/user1', 'Users/user2')
// and `userId` will be a variable that can be accessed in this `match` body.
match /Users/{userId} {
       // allow only signed in users to read the document.
       allow read: if request.auth != null;

       // allow only the user that his/her UID matches `userId` (which is the document's id)
       // and the 'name' field must be a string.
       allow write: if request.auth.uid == uid && request.resource.name is string.
}
```

4.2.2.3 Firebase Cloud Functions

Cloud Functions is a Firebase feature that allows the system to run backend code in the cloud on different events called either by HTTPS requests or one of Firebase's several triggers. Most of the functions deployed in this system are triggered by Firestore triggers, which run the functions when a document gets created, updated, or deleted.

For the rest of this report, cloud functions names will be prefixed with “CF:”. So, a cloud function with the name “hello” will be written as “`CF:hello`”.

The deployed functions are:

- `CF:toolCreated` runs when a new document is created in the ‘Tools’ collection. Its job is to send the new tool’s data to Algolia.
- `CF:toolUpdated` runs when a tool document is updated, and it handles accepting a request by checking if the field `acceptedRequestID` was changed.
- `CF:toolDeleted` runs when a tool document is deleted. It deletes the tool’s data from Algolia, deletes the tool pictures and videos from storage, and deletes the ‘requests’ subcollection.
- `CF:requestWrite` runs when a tool-request document gets created, updated, or deleted. It handles sending notifications, creating a reference in the renter’s ‘requests’ subcollection, and handling the request if it was accepted before being deleted.
- `CF:IdCreated` runs when the user adds his/her id number. It adds the ID number to the ‘idsList’ collection and updates the user’s checklist document.
- `CF:deliverMeetingUpdated` runs when the delivery meeting document gets updated. It handles the flow of the meeting. For example, if a user sets his/her arrival to false all the next steps will be set to false which are media and IDs confirmation in this case. It also adds the users’ IDs in the meeting document when the ID confirmation step comes. And it starts the payment process and rent after IDs confirmation.
- `CF:dMeetPaymendDocUpdated` runs when the ‘private/payments_processing’ document of the delivery meeting gets updated. It handles the payment flow and verification for starting the rent.
- `CF:returnMeetingUpdated` runs when return meeting document gets updated. It handles the flow of the meeting and the payments requests and refunds at the end of the rent.
- `CF:disagreementCaseUpdated` runs when a disagreement case document gets updated. When a case is reviewed and a result is submitted, it sends a notification to the renter and owner of the decision and updates the return meeting document.
- `CF:addSourceFromToken` is an HTTPS callable function. It’s called when the user has a token for his/her credit/debit card and wants to add it to the system. The function will then call Checkout and verify the card. This will be explained further in [4.4.2 Adding a credit/debit card].
- `CF:deleteCard` is also HTTPS callable, and it’s called when the user wants to remove his/her credit/debit card from the system.

- `CF:payments` is also an HTTPS callable function. However, it's not meant to be called by users and it'll return a 401- Unauthorized error, but it's used as a webhook endpoint between Checkout and the system. This function will be explained further in [4.4.1 Webhook].
- `CF:newUser` runs when a new user account is created and creates the user's document in the 'Users' collection.
- `CF:updateUsername` is an HTTPS function called by users to update their username.
- `CF:updateUserPhoto` is an HTTPS function called by users to update their profile photos.
- `CF:banUser` is an HTTPS function callable only by an admin and it's used to ban a user. It disables the user's account in Firebase Authentication and adds the user's ID number and UID in the 'bannedList' and 'bannedUsers' collections respectively.
- `CF:reviewWrite` runs when a user review gets created, updated, or deleted and it updates the user's rating.
- `CF:newNotifications` runs when a new notification document is created and calls Firebase Cloud Messaging (FCM) API to send the notification to the user's devices.
- `CF:adminChange` runs when a document is created, updated, or deleted in the 'admins' collection. When a document is created it takes the document ID (which is a UID) and sets the user with this UID as an admin. And the opposite if the document is deleted, where this function will unassign the user from the admin role.

4.2.2.4 Push Notifications

Firebase Cloud Messaging (FCM) is what's used to send push notifications to users' devices. As its documentation defines it, FCM "is a cross-platform messaging solution that lets you reliably send messages at no cost." (Firebase Docs, 2021)

Notifications will be explained further in [4.5 Notifications].

4.2.2.5 Files Storage

As for the system storage, *Firebase Storage* is used to store all the pictures, videos, and any other files that need to be stored. The system storage is split into two *buckets*, which are like containers where the data is held.

The first bucket is the default one, which has four folders, `'tools_media'`, `'deliver_meetings'`, `'return_meetings'`, and `'userPhotos'`.

- `'tools_media'` stores the photos and videos of the tools' posts. Each tool has its media stored in a subfolder with the folder name being the tool ID.
- `'deliver_meetings'` and `'return_meetings'` store the media of delivery meetings and return meetings respectively. The media is stored in a folder with the tool ID and a subfolder with the request ID. So, the media for the delivery meeting of the request "`req_123`" on tool "`t_65`" are stored in `"deliver_meetings/t_65/req_123"`.

- `'userPhotos'` stores the users' profile photos (avatars) with the file name being the user's UID.

The second bucket is "rentool-terms" which stores the system's Terms of Use (TOS) and Privacy Policy. And this bucket is CORS configured.

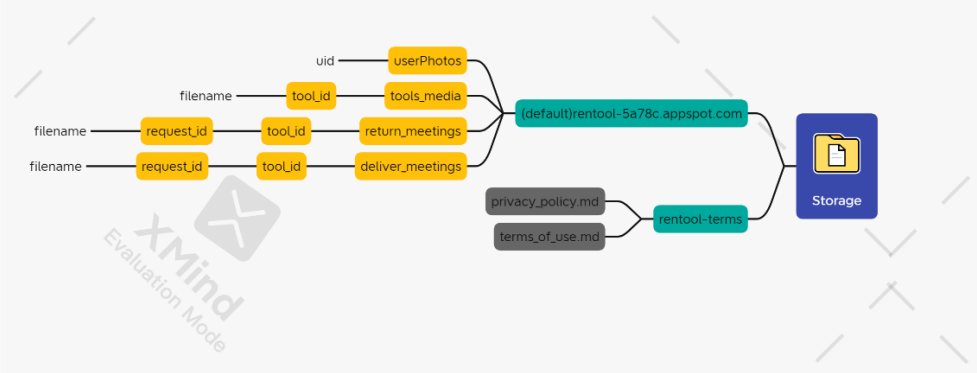

Figure 10 A simplified map of the Storage buckets. Colors: green: bucket, yellow: folder, black: specific file.

4.2.2.6 Hosting

Firebase Hosting is what's used to host the web app. It hosts the app on https://rentool.site, and the other two free domains https://rentool-5a78c.firebaseapp.com and https://rentool-5a78c.web.app.

4.2.2.7 Searching

Algolia is a dedicated search service that is used in this system to search the tool posts. Algolia offers a better experience and results compared to using Firestore queries. This will be expanded on in section [4.8 Search].

4.4 Payments

As mentioned in [2.1 Payments], the payment gateway I decided to use is *Checkout*.

4.4.1 Webhook

Checkout allows developers to create a webhook where the developer provides a URL that checkout will send details of chosen events to. The chosen events decided for this system are Card verified, Payment canceled, Payment declined, Payment paid, Payment captured, Payment approved, Payment capture declined, and Payment expired. `CF:payments` is deployed as an HTTPS function on the URL https://rentool.site/payments, and it's written to handle Checkout's webhook events. Whenever one of those events happens, Checkout will send an HTTPS request with the details to `CF:payments` which will handle it appropriately. To ensure that the request received is from Checkout and not a malicious user, Checkout sends a secret key in the request's headers. So, when `CF:payments` receives a request, it checks its headers for this secret key.

4.4.2 Adding a credit/debit card

For a user to create a post or send a tool-request he/she must have a card associated with them. As this system is not PCI-certified, the system cannot store users' card details and it must be stored in the payment gateway's servers instead.

To add a card the user would go to the payment settings page in the app and fill in the card details. Then, the app (client-side) would call *Checkout's* API and send it the card details to generate a token for the card. Then the app would send the token to `CF:addSourceFromToken` which will then use the token to call *Checkout*'s API and request a card verification (which is done by requesting a payment with the amount of 0). Checkout will then contact the card's bank to verify it and associate it with a *Customer* object with the user's email address. Checkout will then send back the verification status (i.e., "Declined", "Card Verified" or "Pending") and sharable card details (e.g., last 4 numbers, bin, card type), which if the card was not declined will then be added and the user's checklist document ("`Users/{user's uid}/private/checklist`") will be updated. If the status was "Pending" it means the user must authorize the operation with 3D Secure. Which after he/she does that. Checkout will send a webhook event to `CF:payments`, which will update the database.

The response returned from Checkout contains the field 'payouts' which tells if the card accepts payouts (i.e., money can be sent directly **to** the card). If the card does support payouts, then the user can create tool posts. Otherwise, the user will only be able to send tool-requests (i.e., only able to rent other users' tools).

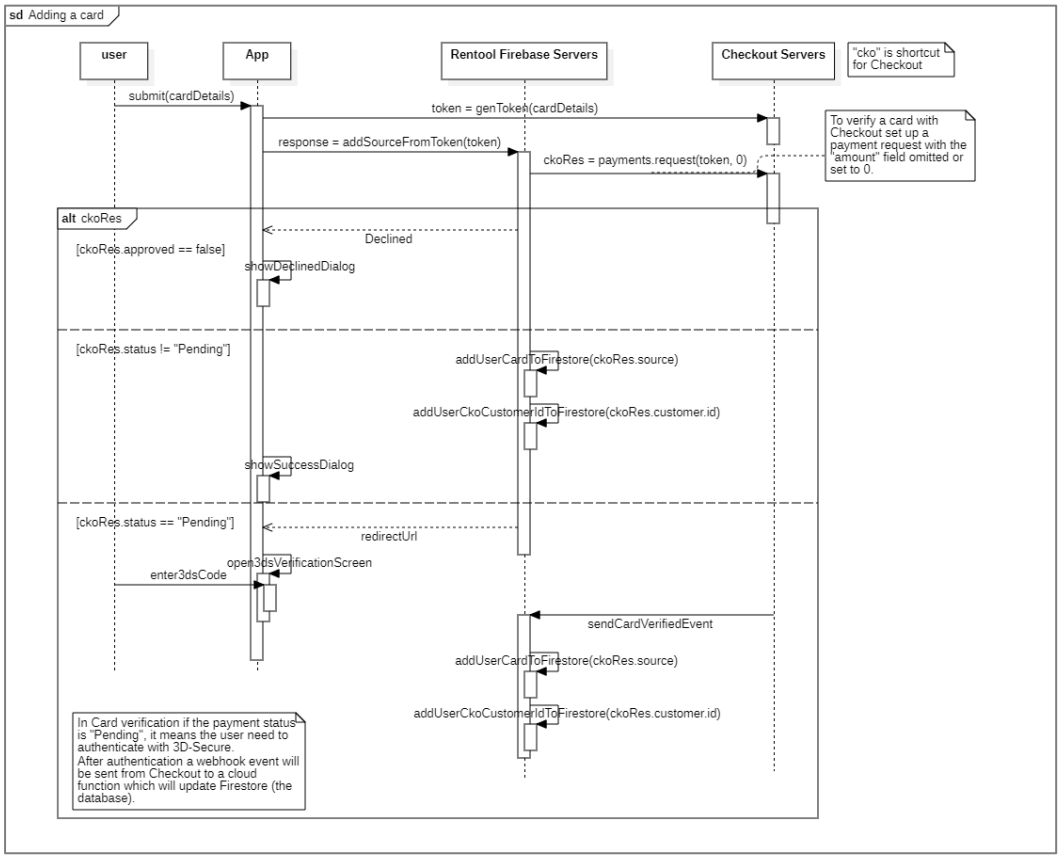

Figure 11 Sequence diagram of adding a credit/debit card

4.4.3 Handling start and end rent payments

4.4.3.1 Delivery meetings

The `payments_processing` document of the 'private' subcollection of the delivery meeting document (path = `"{deliver_meeting_document}/private/payments_processing"`) stores information about the payments flow of starting the rent. It has the following fields which all default to false:

- `'renter_sent_charge'` turns to true when Checkout's API gets called and the renter payment gets requested.
- `'renter_paid'` turns to true when the renter payment gets captured (i.e., the renter was charged and the money is transferred).
- `'owner_sent_payment'` turns to true when Checkout's API gets called and the owner payout gets requested.
- `'owner_paid'` turns to true when the owner is paid the rent price.

When both the owner and renter confirm each other's IDs, `CF:deliverMeetingUpdated` who is listening to changes in the delivery meeting document will call Checkout API and requests a payment in the amount of the rent price + the insurance deposit amount from the renter. Once the payment is captured Checkout will send a webhook event to `CF:payments` which will update `renter_paid` to true. This will trigger `CF:dMeetPaymendDocUpdated` which will send a payout of 95% of the rent price to the owner, deducting a 5% fee. Once the payment is sent, Checkout will send a webhook event to `CF:payments` which will handle it and update `owner_paid` to true. This will trigger `CF:dMeetPaymendDocUpdated` again which will change the `payments_successful` field of the meeting document to true which will make `CF:deliverMeetingUpdated` starts the rent.

4.4.3.2 Return meetings

For the return meetings `CF:returnMeetingUpdated` will listen and detect once the owner and renter confirm hand-over and it will call Checkout API requesting a refund to the renter of the insurance amount if the tool was not damaged, and the insurance amount – compensation price if the tool was damaged. And then send a payout to the owner with the compensation price if the tool was damaged.

4.5 Notifications

The user notifications are stored in the '`notifications`' subcollection of the user's user document ('`Users/{uid}/notifications`'). Whenever a new document is created in this subcollection, `CF:newNotification` gets triggered and sends a message to the user's devices using Firebase Cloud Messaging (FCM). FCM uses tokens to identify the devices, which are stored in the 'devices' subcollection of the user's user document. These tokens are automatically added to the subcollection when the user signs in where the app would generate a token for the device and then store it in the subcollection. And the token gets removed when the user signs out or when they turn the notifications off on that device.

The notifications are stored as codes. For example, "REQ_REC" for 'request received'. And any extra data is also stored with it (e.g., the tool name). This is done so it can be

displayed in any language in the app by taking the code and data and displaying its explanation text in the user's chosen language.

4.6 Posts and requests

4.6.1 Creating posts

To create a tool post the user must have a verified email address, set an ID number, and enter a credit/debit card that supports payouts. If the user has done all of that, then they can create a tool by filling the tool details in the new post screen and when they tap the "create" button a document in the 'Tools' collection will be created, and its data will be sent to Algolia to be searchable by `CF:toolCreated`.

4.6.2 Sending a request

Similar to creating a post, for the user to be able to send a request he/she must have a verified email address, set an ID number, and enter a credit/debit card (doesn't need to support payouts though). After that, the user can go to a tool's post page and press the "request" button. Fill in the request details and press "send". This will create a new document in the tool's 'requests' subcollection, triggering `CF:requestWrite` which will send a notification to the owner and create a reference document in the 'requests' subcollection of the renter's user document.

4.6.3 Accepting requests

For a tool-request to be accepted, the tool owner would change the tool's `acceptedRequestId` to the request ID he/she wishes to accept. This will trigger `CF:toolUpdated` which will create a delivery meeting document and the 'messages' subcollection for chatting (this will be explained next in [4.6.4 Chat System]), update the request's `isAccepted` field to `true`, and create a notification document in the owner's notifications collection (which as described in [4.5 Notifications], will send a notification to the owner).

4.6.4 Chat System

When a tool request is accepted, the 'messages' subcollection is created and it contains documents of messages between the owner and renter. When a user sends a message a document is created containing the message, the time it was sent, and the user's UID. On the app, the chat page is listening to changes in the 'messages' subcollection and it populates and sort the message documents as they are added to the subcollection. This is possible by Firestore listeners that allow the app to listen for changes in a collection or a document. So, the app would create a listener and build the messages using a Flutter widget called StreamBuilder. This widget listens to a stream and rebuilds itself whenever any message gets sent through the stream.

4.7 Reviews

Once a rent ends, the document with the owner's UID as its ID in the `previous_users` subcollection of the renter's user document will be updated to have a key-value field

of `{tool id}: true`, and vice versa for the owner. This is how the system knows if two users had a previous rent between them (FR23). If a user's UID is in another user's `previous_users` subcollection, then the two users can post a review of each other. For example, if the renter with UID "user_11" and the owner with UID "user_22" finished rent for the tool with ID "tool_232", The following changes would happen:

- The document '`Users/user_11/previous_users/user_22`' will be updated to have the field "`tool_232`" set to `true`.
- The document '`Users/user_22/previous_users/user_11`' will be updated to have the field "`tool_232`" set to `true`.

The reviews are created in the reviewed user's `reviews` subcollection ('`Users/{reviewed user's UID}/reviews`'). The review must have a value of 1 to 5 and an optional description of up to 500 characters. Once the user posts the review, `CF:reviewWrite` which listens to the `reviews` subcollection will get triggered and will update the reviewed user's document's `rating` and `numOfReviews` fields.

4.8 Search

Firestore is designed for fast and simple queries, and to keep the high-performance, Firestore doesn't support native indexing or search for text fields in documents, which makes full text searching hard and a bad experience and it's not recommended by Firestore documentation. For a better full-text search, Firestore documentation suggests using a dedicated third-party search service such as Elastic, Algolia, or Typesense. After some research, I decided to choose Algolia, as it has a free pricing plan for the first 10k queries and a lot of resources and guides on how to implement it with Firestore and Flutter.

So, the way it works is that Algolia stores the tools' data in an *index* where every tool is stored as a record, and then when the user searches for a tool, a request is sent to Algolia's API where it searches through the data and returns the most relevant results. The data is stored in an Algolia index with the name "tools". The way this data is populated is using three cloud functions: `CF:toolCreated`, `CF:toolUpdated`, and `CF:toolDeleted`. `CF:toolCreated` gets triggered when a new tool-post is created and sends its data to Algolia's servers. `CF:toolUpdated` gets triggered when a tool-post gets edited and sends the changes to Algolia. `CF:toolCreated` gets triggered when a user deletes a tool-post and sends a request to Algolia to delete its data. The tools' IDs are the same in Algolia and Firestore which makes the search and link process seamless.

However, when the system is running locally (on Firebase local emulators) the Algolia index that will be used is "tools_test". This is necessary to separate the production environment and the testing and development environment.

4.9 Admins

The system admins' roles are split into two groups, Firebase admins and admin users. Firebase admins are admins that don't have to have an account in the system (a Rentool account), but they need a Google account. These admins are assigned access to the Firebase console and can be invited in the project settings page. A Firebase admin can be an editor or a viewer, either to all Firebase services or to a chosen set of them.

An admin user is a user that has an account with the custom claim `'admin:true'`. Custom claims are custom attributes on user accounts. The admin custom claim can be added to a user by a Firebase admin by creating a new document with the user's UID as its ID in the 'admins' collection which will trigger `CF:adminChange` and sets the user as an admin.

In this report, "admin" refers to admin users (the ones with accounts), and "*Editor*" and "*Viewer*" (in italics) refer to Firebase admins.

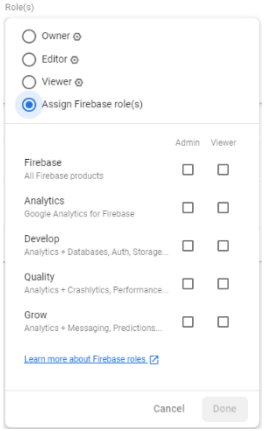

Figure 12 The menu for setting a Firebase admin's role.

4.9.1 Disagreement cases

When an owner claims the tool is damaged and the renter says he/she didn't damage it, the app will prompt them to take picture and videos of the tool and submit it. Once they do, the return meeting will be locked, and `CF:returnMeetingUpdated` creates a disagreement case in the 'disagreementsCases' collection.

A disagreement case is only resolvable and readable by an admin. The admin opens the admin panel page in the app and can see a list of cases. The admin opens a case then looks at the pictures and videos of the tool before the rent (taken in the delivery meeting) and after. Then, the admin enters the decision and the reasoning behind it and submits it. Once submitted `CF:disagreementCaseUpdated` sends a notification to the owner and renter and updates the return meeting document.

4.10 Testing

Testing is automated using a group of unit tests and integration tests. The tests are run locally using Firebase local emulators. The unit tests don't require the app to be compiled or launched since it runs by directly calling Firebase local APIs and testing the response. Integration tests however require the app to be launched on a connected device where a series of taps, entering texts, and other user interactions will be simulated to test if the app is working correctly.

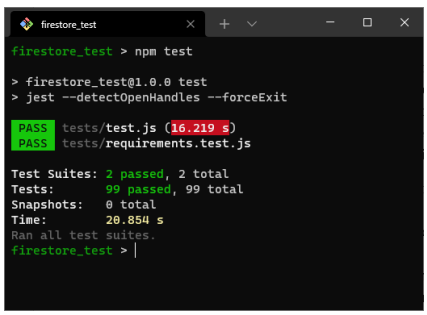

Figure 13 The output of a successful run of Firestore unit tests.

All the tests can be run at once by the batch script in the main folder, 'test.sh' by running the command "`sh test.sh`". This command will start Firebase local emulators, run a setup script that will create some

accounts like an admin account and an account with a verified email address. Then it'll check to see if there are any android devices connected and start the integration tests if there are any. Then it'll start the unit tests. And finally, after all the tests are finished, it'll shut down the emulators.

The unit tests can be used and launched manually at any time by the developer. But they're also set to run automatically when there is a pull-request on the master branch in the system's repository on GitHub. This is done through GitHub Actions and is configured in the ".github\workflows\ci.yml" file in the repository.

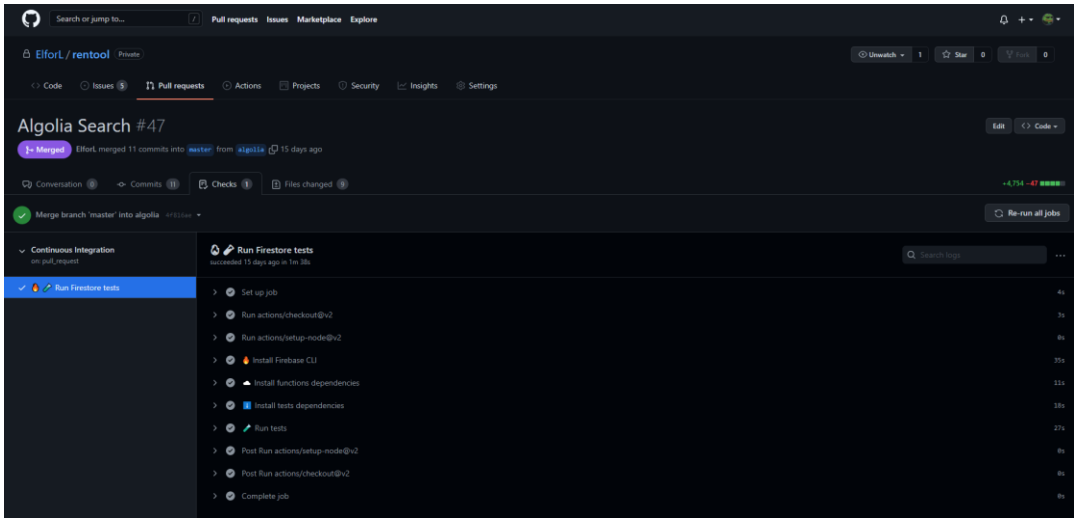

Figure 14 Successful tests with GitHub Actions on pull request #47. This pull request was used to merge searching with Algolia feature to the master branch.

4.11 Structure of Implementation-Folder

The main project folder contains all the code for the app and cloud functions, the security rules, hosting configuration…etc. Most of the files and folders in at are for the app (Flutter files). But it also contains the Firebase files including cloud functions code, Firestore and Firebase Storage security rules, hosting configuration, and others. Source control is also implemented on the folder using Git and uploaded to GitHub on the repository ElforL/rentool.

This is a list of the important folders and files and their uses:

- **'.github' folder**
  - This folder contains GitHub actions configuration. Currently, it's configured to run the tests automatically when there's a pull request on the master branch (continues integration).
- **'android' folder**
  - contains the configuration files for the android app like the android manifest, Gradle configurations, the app icon…etc.
- **'assets' folder**
  - contains the app assets including images like the app logo, and fonts used in the app.
- **'FirebaseStorage' folder**
  - contains the security rules for Firebase Storage.

- **‘Firestore’ folder**
  - contains the security rules and indexes of Firestore.
- **‘functions’ folder**
  - contains a node.js project implementing the cloud functions written in TypeScript.
- **‘build’ folder**
  - Contains the built app files. After building the app (using the command `flutter build`) the files will be in this folder.
- **‘ios’ folder**
  - contains the configuration files for the iOS app.
- **‘lib’ folder**
  - contains the main code for the Flutter app.
- **‘integration_test’ folder**
  - contains Flutter integration tests files.
- **‘test’ folder**
  - contains the regular Flutter widget tests and Firestore unit tests.
- **‘web’ folder**
  - contains the configuration files for the web app.
- **‘firebase.json’**
  - The configuration file for all Firebase files in the main folder. It also contains the hosting configuration.
- **‘l10n.yaml’**
  - Configuration file for localization (showing the app in multiple languages)
- **‘pubspec.yaml’**
  - Manages the app dependencies, assets, and version.
- **‘test.sh’**
  - a batch script that runs Firebase local emulators and runs the tests.

Chapter 5 – Results and discussion

The biggest skill I learned in this project was implementing a payment solution to the system. This project is the first project I ever implemented a payment gateway with. It was a difficult experience to find the perfect one and learn how to implement it and the best practices for requesting payments and verifying them. But eventually, a payment solution was implemented successfully using Checkout and their amazing payout feature which allowed for a very simple experience for the tools owners to get paid without them having to create a special account and entering their bank details like other e-commerce platforms. Another small thing I learned and improved in was in Flutter (the app/user interface), and it was using the `FutureBuilder` widget for building a responsive UI for asynchronous operations.

Overall, the project goals were achieved successfully. The system is a fully functional rental platform. Users can create accounts and sign into them easily with no problems. The system allows users to offer their tools. It allows users to rent other users' tools. It implements multiple features to ensure the users' rights using disagreement cases and pictures and videos before and after the rent.

Two functional requirements were not completely finished, and they are FR3 and FR35.

FR3 (The system must allow the user to log-in/sign-up using a Google, Facebook, Microsoft, or Apple account.) was not **entirely** completed. The system does support signing in with Google and Facebook, but not with Apple and Microsoft. The reason is that Apple requires an unreasonable $99 annual fee for a developer account which I can't pay. As for Microsoft, I could not find a way to implement it with Flutter. I was only able to make it work on the web app, but it always failed in the native apps (Android and iOS). So, I decided to not waste any more time working on these two features and continued working on the other requirements. Another future option that may compensate for this is adding signing in with other providers that have supported implementation with Flutter, such as Twitter and GitHub.

FR35 (The system must allow the user to upload documents) was canceled due to improper planning during the analysis phase. After some thought, I decided to delay it until I develop a page specifically for users to contact the admins, where the user would only be allowed to upload a file if there was a reference id that the system will use to determine if he/she allowed to upload. This will prevent users from uploading documents at any time and filling the system's storage. For now, the user will upload any required documents using email while he/she is contacting the support team.

There are multiple other features that I would like to work on and implement to improve the system. For example, increasing the system's income. Right now, the only income is the 5% deduction from the rent price. An option that I would like to implement is ads. Ads are a popular option and are always a great source of income. Google ads are the most popular ads used in multiple websites and apps and implementing them should be very easy with this system since there are official packages and libraries by Google to implement them with Firebase and Flutter. Firebase with AdMob, and Flutter with 'google_mobile_ads' package, and the new ads support announced with Flutter 2.0.

Another option is sponsored tool posts where a user pays for his/her tool to be shown first in search results with an “AD” sign on it. This is the same as Google sponsored results, where paid options are shown first (like the image below).

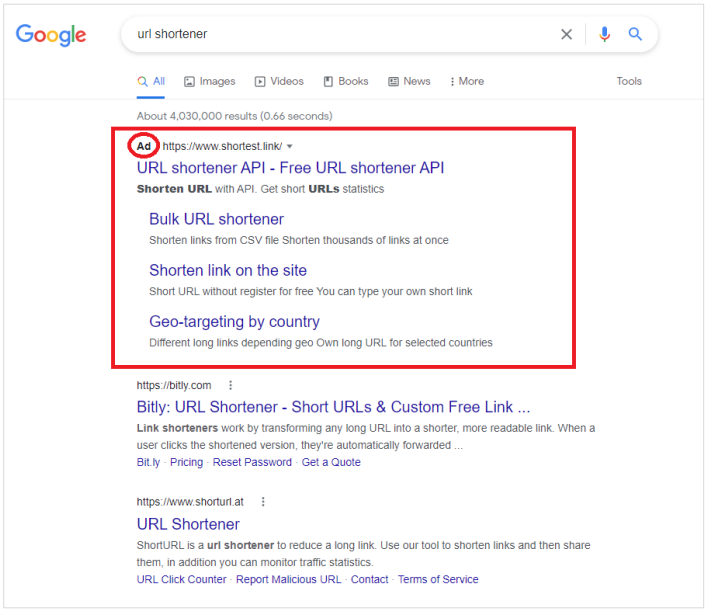

Figure 15 Google sponsored results. The first result's website paid Google to be shown first.

Post suggestions are another feature that I didn’t have time to research and implement. It may be achievable using Algolia *Recommend* feature or using Google Analytics to see what other tools users rented and use them as recommendations.

Another great but very hard feature would be *delivery*. Where the system will contact a delivery company like Aramex or UPS, for them to receive the tool from the renter and deliver it to the owner. This of course means rethinking the delivery and return meetings and a bunch of other system operations, but it’ll help greatly grow the system’s user base.

Chapter 6 – Conclusion

From 3rd February 2021 to 6th December 2021, I have researched, designed, and built this system called "Rentool" which achieved its goals of providing an online platform for users to create an account where they can offer their tools for rent and to rent other users' tools.

The development model used was the incremental and iterative model. Chosen due to missing requirements due to the lack of knowledge and experience in building e–commerce and rental platforms.

The system is deployed on three apps, an Android app, an iOS app, and a web app hosted on https://rentool.site. The system's backend is deployed with Firebase, with Firestore as the database, Cloud functions for running backend code which gets triggered by multiple triggers including changes in the database. or the creation of a new user account. or by calling them directly. Cloud Storage for storing the pictures videos and any other files the system needs. Firebase Cloud Messaging for sending push notifications to users' devices. And Firebase Hosting for hosting the web app. The system allows the user to search for a tool. This is done using Algolia where Cloud functions would detect creation, update, and deletion of tools and update Algolia.

To offer a tool the user would have to create an account, verify his/her email address, set his/her national ID number, and add a credit/debit card that supports payouts. Then the user can create a tool document in the 'Tools' collection where users who also had set an ID number and added a credit/debit card can send requests to. Then the owner would review these requests where he can reject them or accept one of them. after that the owner and renter would meet to deliver the tool and after the renter is done, they would meet again to return the tool. Here the owner can claim the tool is damaged and create a disagreement case that an admin would have to review and decide if it was indeed damaged or not.

The system handles payments between the owners and renters using *Checkout*. Checkout is a payment gateway that allows money to be sent from and to users' credit and debit cards. The system has a webhook with Checkout servers using `CF:payments` which listens to events sent from Checkout like a payment being captured or declined and handles it appropriately. To keep the system up and profitable, the system takes 5% of the rent price as a fee.

The system can be tested using a set of unit tests and integrations tests to see if it’s working correctly. These tests also run automatically when there’s a pull-request on the master branch in GitHub.

There is more to be achievable with this system. Future work includes additional methods of income such as ads using AdMob or sponsored search results, post suggestions, and researching a delivery feature.

References

Checkout, 2021. *Checkout.com - Docs.* [Online]
Available at: https://www.checkout.com/docs
[Accessed 1 12 2021].

Firebase Docs, 2021. *Firebase Cloud Messaging | Firebase Documentation.* [Online]
Available at: https://firebase.google.com/docs/cloud-messaging
[Accessed 5 12 2021].

HSS, 2021. *FAQ - HSS Hire.* [Online]
Available at: https://www.hss.com/hire/faqs
[Accessed 5 April 2021].

Krupadeluxe, 2021. *File:Iterative Process Diagram.svg.* [Online]
Available at: https://commons.wikimedia.org/wiki/File:Iterative_Process_Diagram.svg
[Accessed 7 12 2021].

National Association of REALTORS, 2019. *Remodeling Impact Report: D.I.Y.,* s.l.: National Association of REALTORS.

Rent4Me , 2021. *Rent4Me | Rent, rent out and share anything. Use Sharing Economy. Borrow Cars and Cameras.* [Online]
Available at: https://rent4me.org/
[Accessed 14 4 2021].

Sparetoolz, 2021. *Sparetoolz FAQ.* [Online]
Available at: https://sparetoolz.com/pages/faq
[Accessed 5 April 2021].

Unified National Platform, 2021. *e-Payment channels in the Kingdom of Saudi Arabia.* [Online]
Available at: https://www.my.gov.sa/wps/portal/snp/aboutksa/ePayment
[Accessed 2 December 2021].

United Rentals, 2021. *Equipment Rentals | United Rentals.* [Online]
Available at: https://www.unitedrentals.com/marketplace/equipment#/
[Accessed 5 April 2021].

Appendices

Table of Figures