Revised Design

Requestr

Overview

The purpose of Requestr is to provide a unified platform for MIT students to request help from other students. The motivation for the app stems from the fact that there is no known viable, resourceful forum for MIT students to ask for help. There's piazza for academic help and facebook groups for other circumstances, but they aren't completely effective. For example piazza is class-specific, different people rely on different resources (e.g. office hours), and the purpose of facebook groups is not specific to requests. Requests not pertaining to classes are also a concern. We feel that by creating a central location for general requests and discussion, this app can facilitate stronger interpersonal relationships between students and ultimately build a stronger community.

Requestr provides a consistent way to request help from other users. The users requesting help (requesters) can submit requests to the MIT population. Other users can search for and filter active requests, which are delivered to them in an ordering based on the preferences of each user, then sign up as a helper for a number of requests. These helpers can engage in conversation with the requester and other helpers. Once the request is complete, all users can give feedback to each other, which shows up to other users.

Design Essence

Key Concepts

Request

Purpose: Provides a consistent format users can use to ask for help on the site.

Operational Principle: A user creates a form (via a button) and enters data about the request in fields such as Request Name, Description, Skills Required, Expiration Date, and Reward. Submitting the form then makes the request accessible to other users, who can then apply to help with the request. The requester then has the option to select which users he wants to help with the request.

Categorization

Purpose: Organize requests to allow users to find certain types of requests more easily.

Operational Principle: A user searching for requests can add a filter to find requests pertaining to a specific topic. For example a user looking to practice C++ skills can add C++ as a filter, and will get request results related to C++.

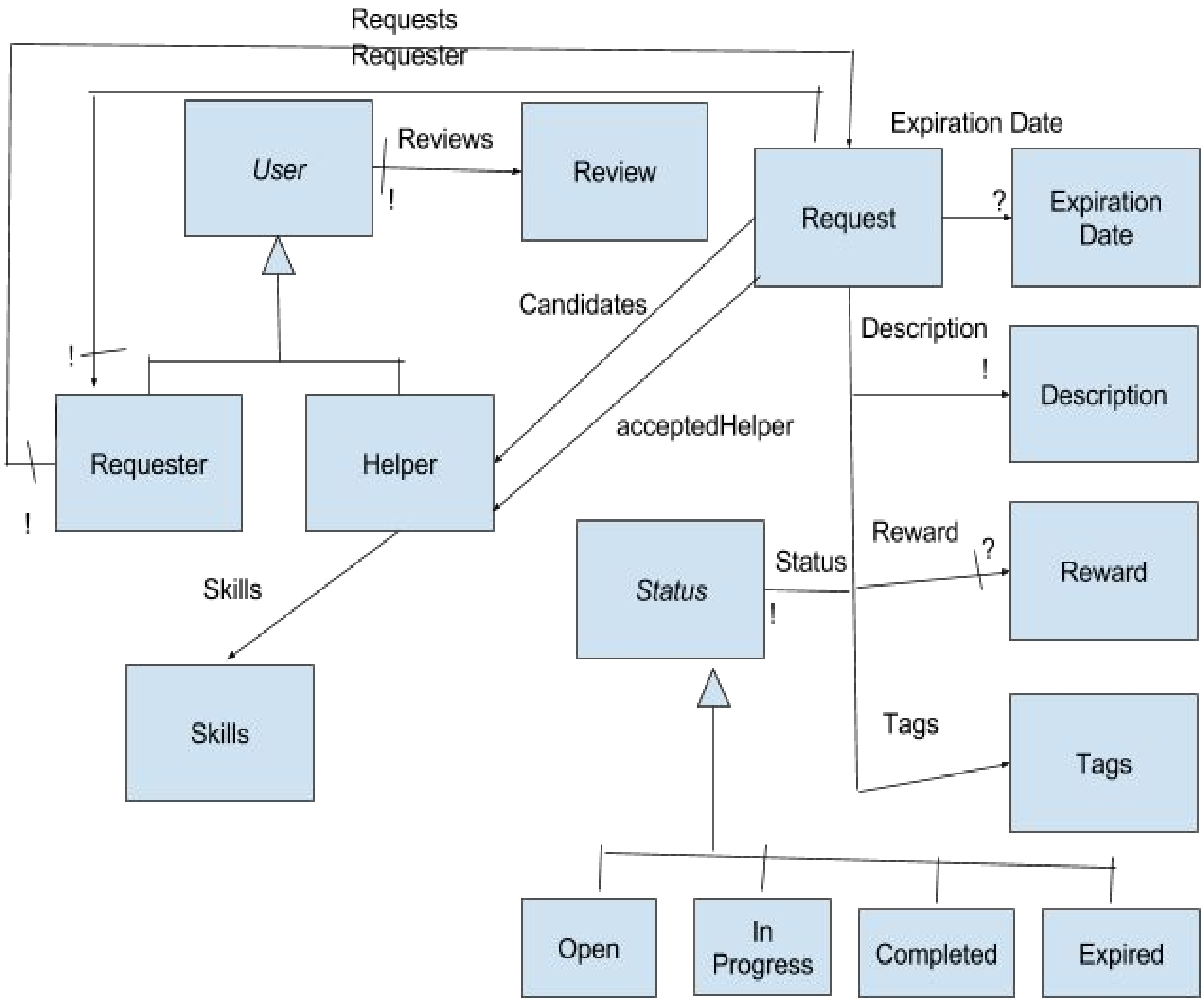
Feedback History

Purpose: Allow users to check on the reliability of other users for fulfilling requests.

Operational Principle: A user puts up a request and other users volunteer to take the request. The requester wants to see how likely it is that each of the volunteers would complete his request the way he wants it done so he looks at the history of feedback each volunteer has received to see how well they accomplished past requests.

Misfit: Users may be able to give others false feedback that will skew other user's views of them when looking at their feedback history.

Data Model



Textual Constraints:

*AcceptedHelper for a request must be one of its Candidates

*A helper for a request cannot be the user that made the request

*A requester can only edit a request that is Open

*If A.requests = m B <-> B.requester = A

Insights:
*There is no strict progression in a request's status. We can have a transition from Open to In Progress and then back to Open if a request is not completed properly by the helpers. The data model currently doesn't reflect what happens if this occurs but an extended design could include a list of blocked users for a request to address this.
*Since the skills of a user can change over time, a user who was not accepted to help a request at one time may be accepted for the same or a similar request in the future.
*After requests expire, there needs to be some way to notify the helpers, since an expired request cannot be completed.

Security Concerns

Threat model: Users can register and login using their email and password. We enforce a check that ensures that only an MIT email is used for registration/login. After registration, an email is sent with a verification link that must be clicked before the account is activated. If users cannot remember their password, they can request for the site to send a recovery email after answering 3 or so security questions that they set up upon registration. Handling requests (the bulk of the app) can only be done when the user is logged in. Account settings also require a valid logged-in user (for obvious reasons).

Who is the attacker: We assume that the attackers, who have malicious intentions, want to compromise as much data as they can. The type of attackers can be anyone that knows of the site. Our main functionality for the app comes from making requests and evaluating users. We do not store any sensitive real-life information besides email accounts. We enforce that much of the app's functionality isn't exposed to anonymous visitors of the site--we only show a login screen for non-logged in users and enforce route-checking where only certain routes/urls (i.e. /createrequest) can only be accessed if the user is logged in. Only people that possess an MIT email address (anyone affiliated with MIT) can register. Thus, the attacker cannot attack the web app from the client side unless the adversary is on the MIT affiliated network (has an MIT email address). This prevents client-sided attacks once logged in. However, the attacker can still be anyone that knows of the site and gains any user's credentials.

What can attackers do: There are two possible scenarios for what happens if our app environment gets hacked.

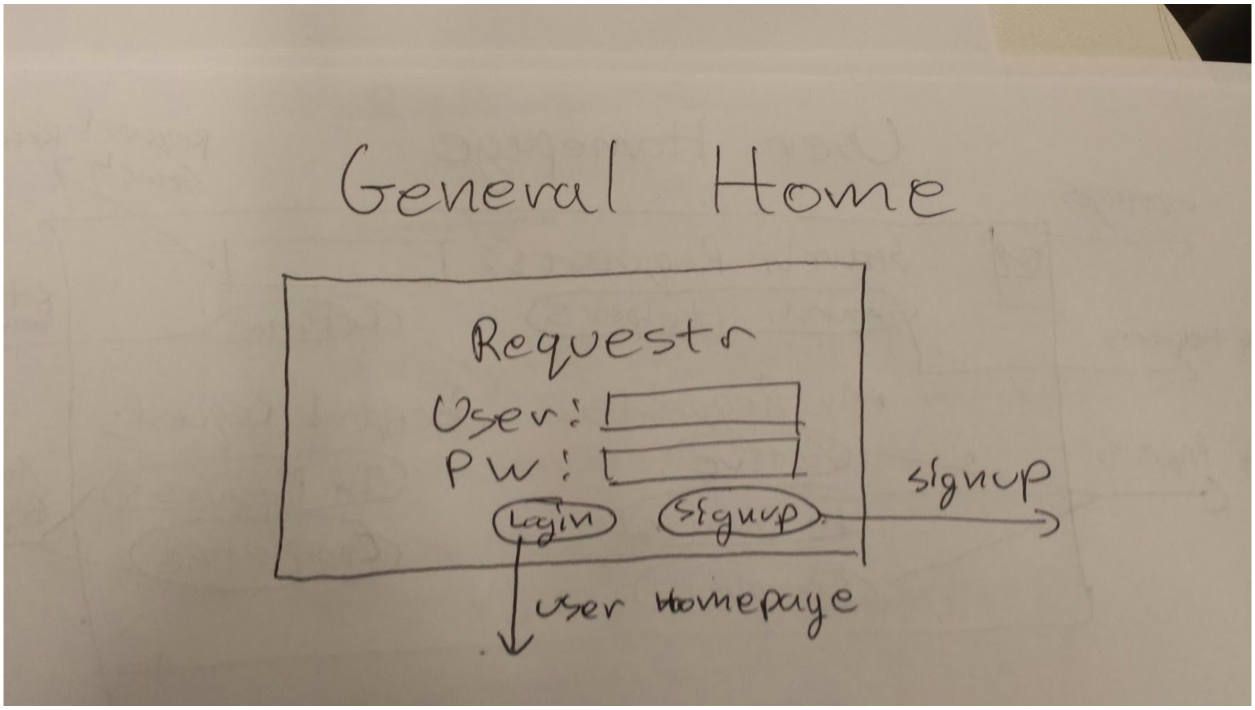
Case 1: An attacker gains access to a client's account. That attacker can then create fake, spurious requests without intentions of rewarding request-takers, take requests himself/herself and not complete them intentionally, jeopardizing trust relationships. Additionally, since we store email accounts on our server, the attacker can possibly send spam emails to the now-known client email.

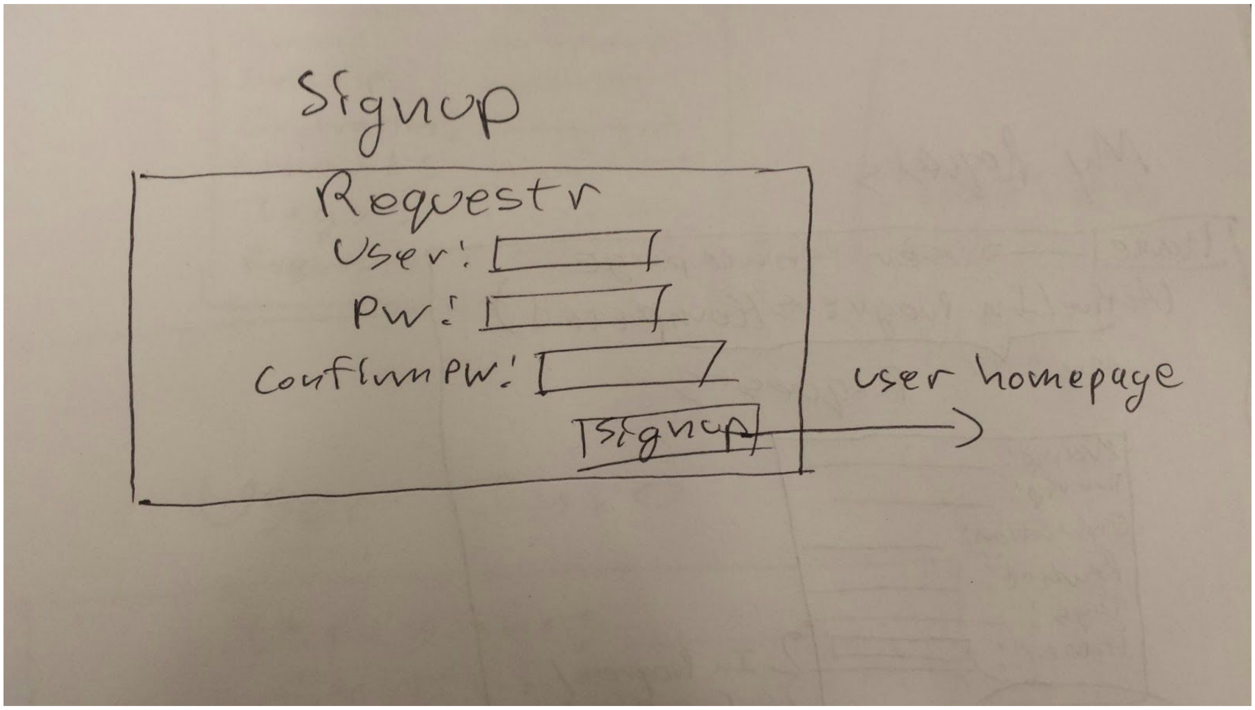
Case 2: An attacker gains access to the server. The server is essentially a conglomerate of multiple users' accounts and relevant data. Thus, when an attacker gains access to the server, the attacker can perform any of the attacks listed in Case 1 as any user, or utilize the list of emails and leverage a spam attack on all of them. If the adversary has control over the backend machine host, he/she can shutdown any or all parts of the server that are on the same compromised host, possibly performing a Denial of Service attack to users.

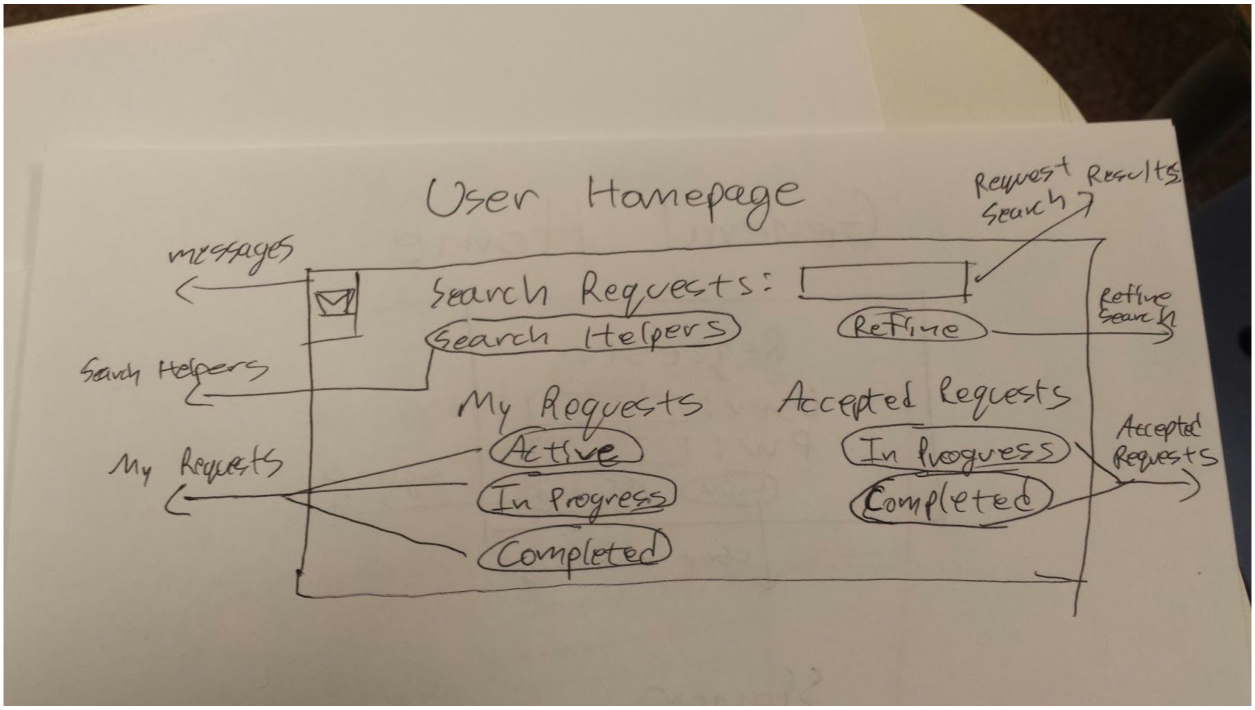
Our web app doesn't operate on any sensitive data transmission, so the highest level of concern a user will have is that someone else will be impersonating their login and/or making/satisfying requests without their permission. Therefore, whatever security holes we forget to address (hopefully none) will not give attackers any leverage on their victims (besides poor reviews/ratings).

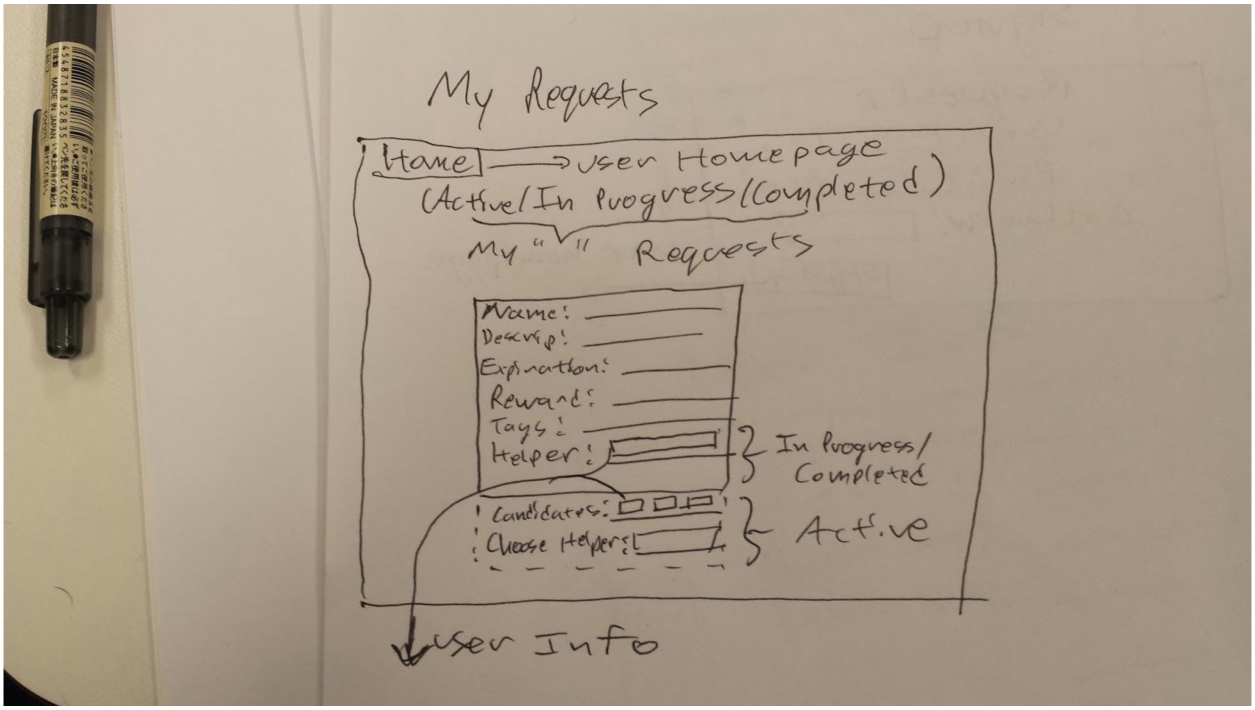
Preventive Measures: Modern platforms can already eliminate much of the web attacks that exist today, such as cross site scripting, sql injection, and cross site request forgery. However, there are still instances where we can't automatically protect from such attacks. There's nothing to prevent any outside attacks that do not utilize the site in any way that allow an attacker to gain access to a client's credentials (i.e. social engineering attacks). However, we hope that the users are sensible beings who wouldn't give away their credentials for the app anywhere else. We can add a message of some sort that reminds users to only input their information when the browser displays the proper URL (to prevent phishing attacks or CSRF attacks). Additionally, we will ensure our forms are sanitized to prevent script injection into any input fields (i.e. by escaping certain characters like quotes and enforcing string-checking to make sure it doesn't contain illegal values such as "<SCRIPT>". That will stop most web attacks such as cross-site-scripting and sql injections. Secondly, we enforce that users who register must provide an email for verification in order to stop spam attacks; we also enforce time-based limits on account registration/login as well as request-taking/making to prevent throttle attacks. To account for cross site request forgery attacks, we will use hidden tokens that are session-specific for each form and check that client requests use the tokens. Denial of Service attacks are out of our control and depend largely on the machine/system security behind the machines that host the servers/backend component of the app. We also implement a design on our app that satisfies separation of concerns (i.e where the authentication server only deals with user accounts and the main application doesn't touch the user database) to avoid authority/control from bleeding over to where it's not needed. We also ensure we log all connections and requests made to and from the server so we can track/trace any potential attacks.

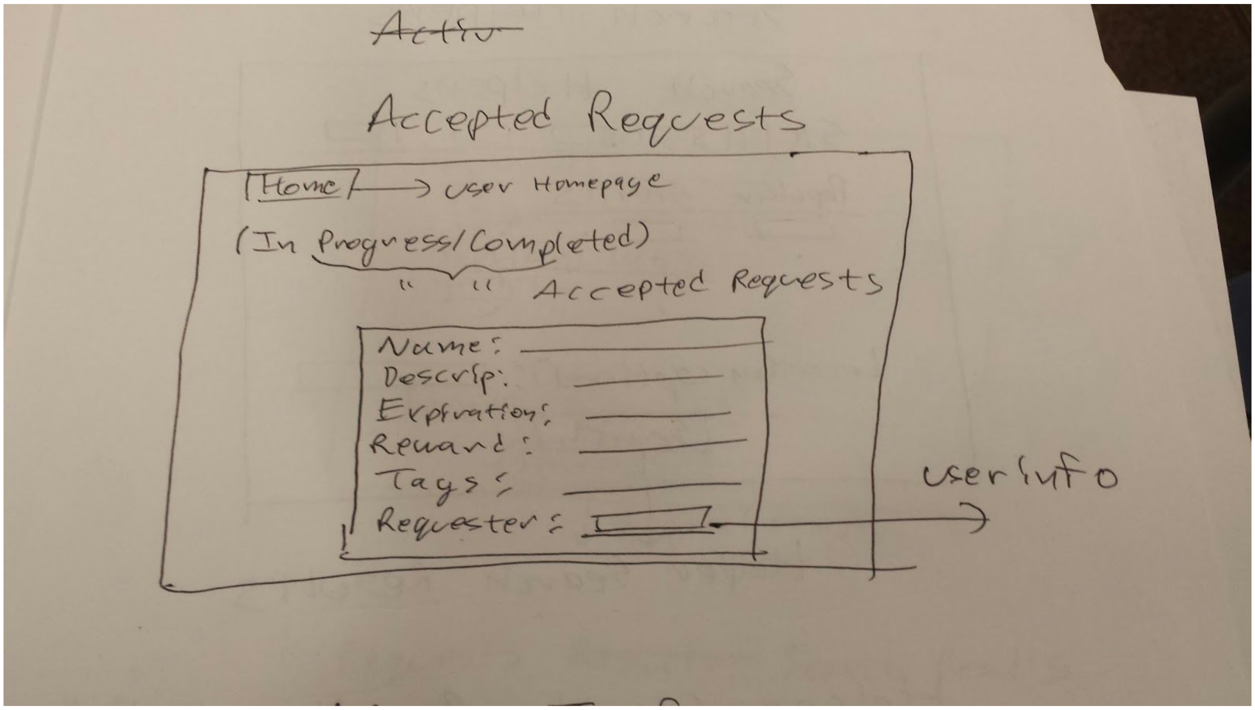
User Interface

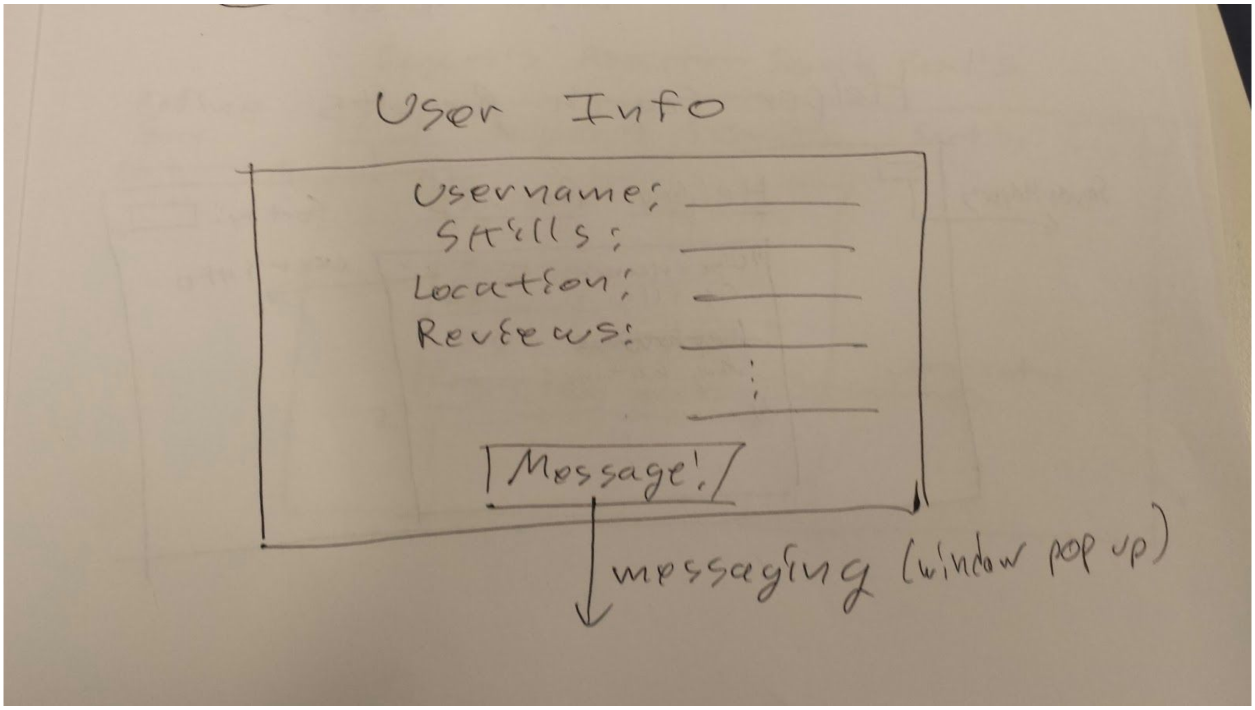


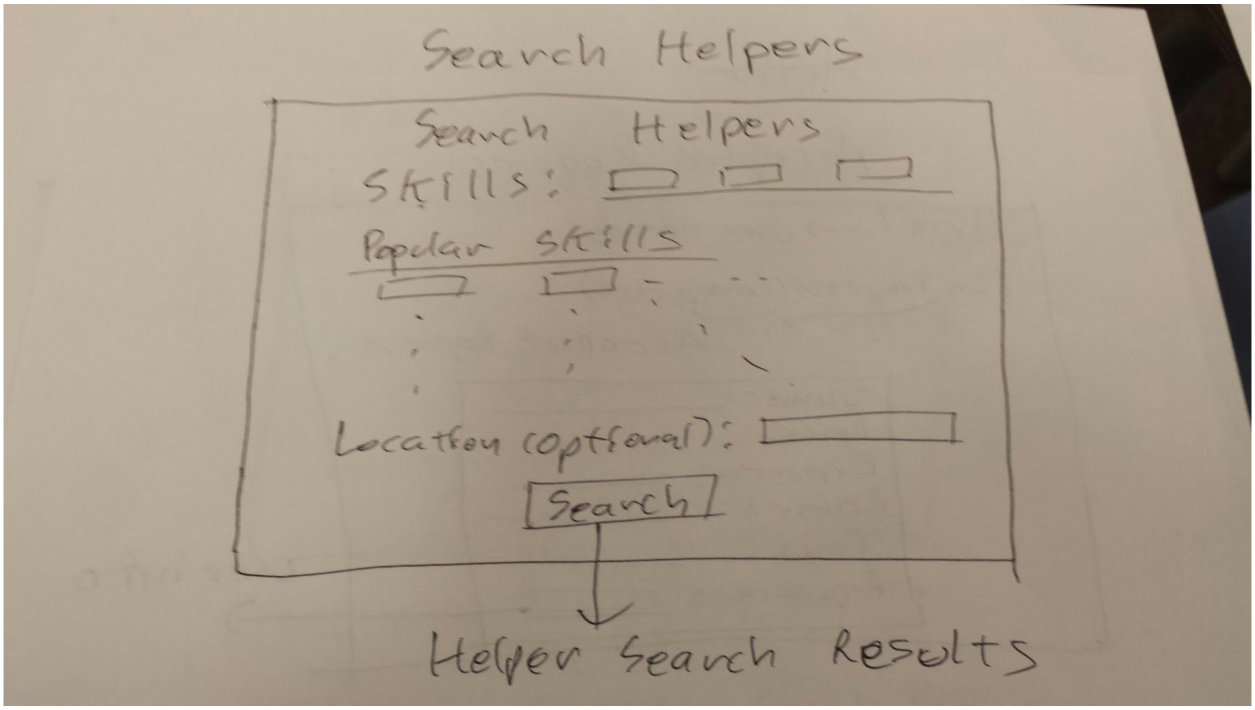


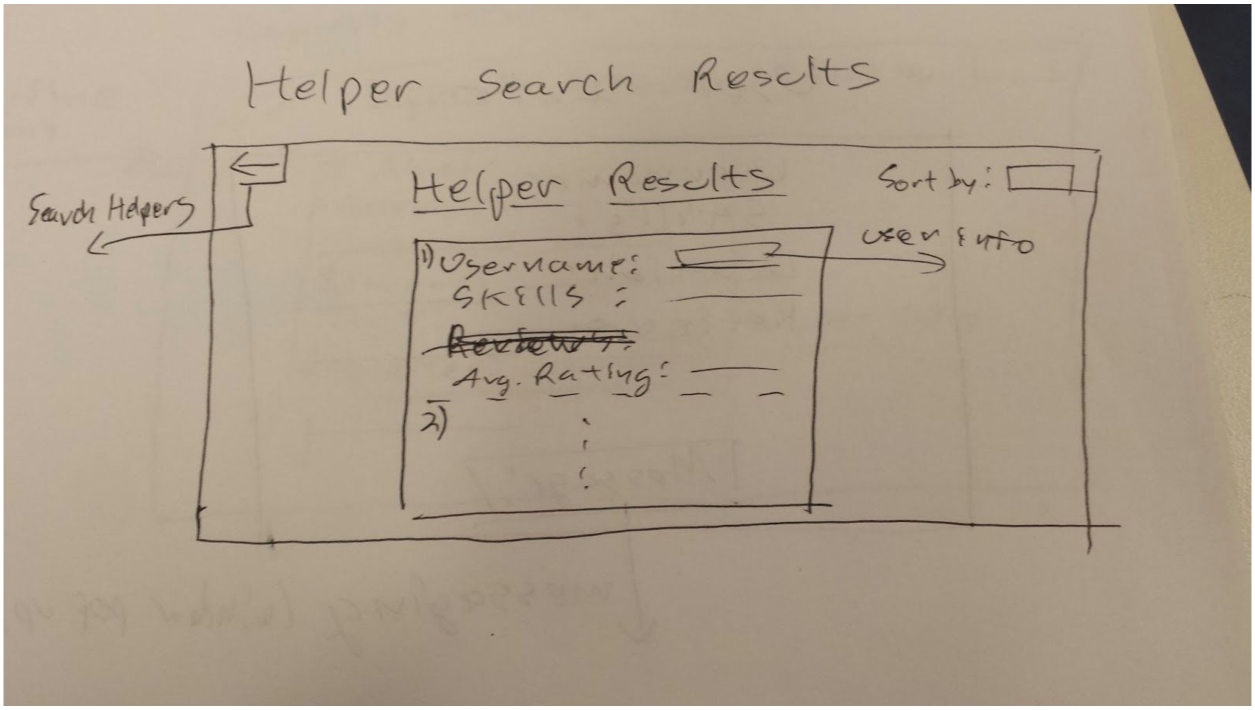

User Homepage
messages
Search Requests:
Request Search
Results
Search Helpers
Refine
Refine Search
Search Helpers
My Requests
Accepted Requests
My Requests
Active
In Progress
Completed
In Progress
Completed
Accepted Requests

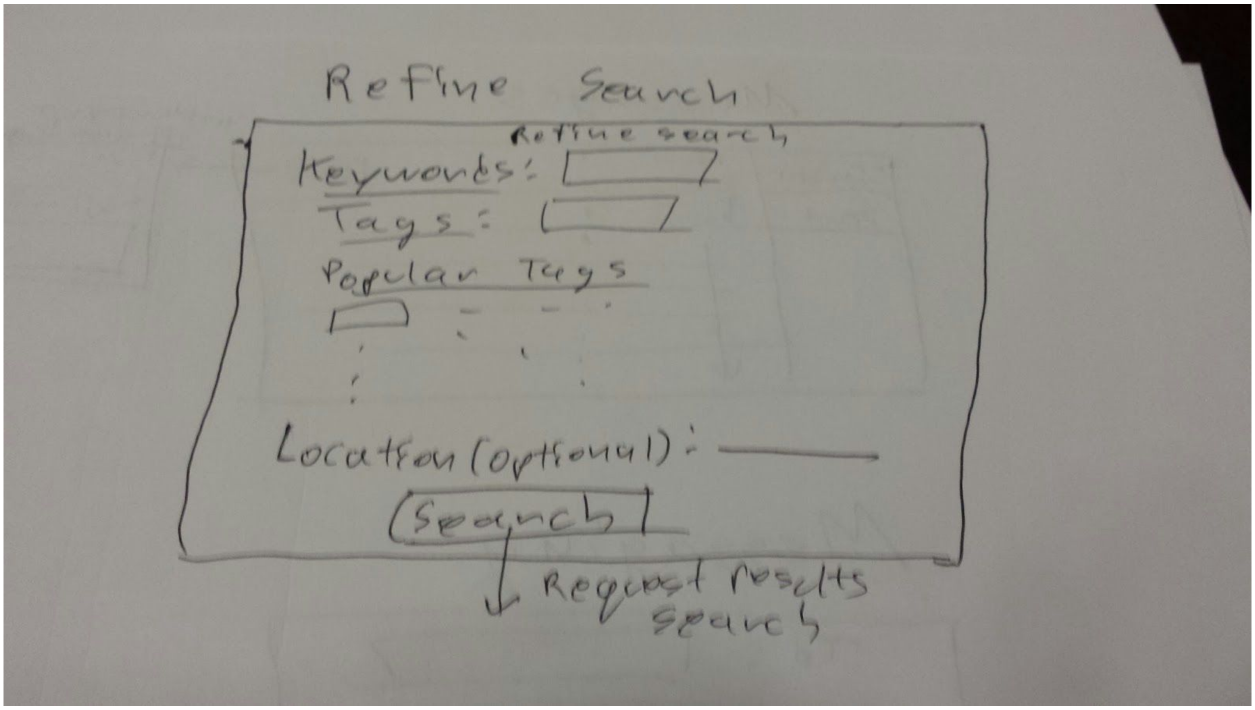

My Requests
Home
User Homepage
(Active/In Progress/Completed)
My " " Requests
Name:
Descrip:
Expiration:
Reward:
Tags:
Helper:
In Progress/ Completed
Candidates:
Choose Helper:
Active
User Info


~~Actio~~
Accepted Requests
Home
user Homepage
(In Progress/Completed)
" " Accepted Requests
Name:
Descrip:
Expiration:
Reward:
Tags:
Requester:
userinfo

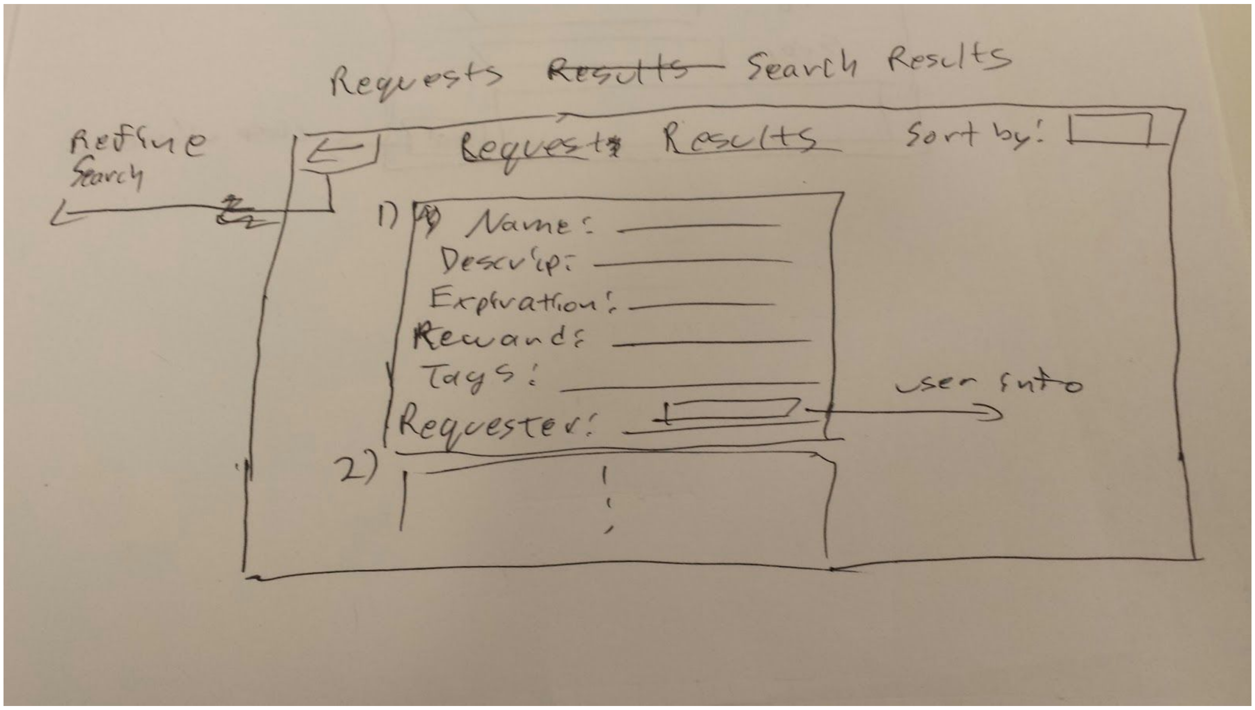

User Info
Username:
Skills:
Location:
Reviews:
Message!
messaging (window pop up)


Search Helpers
Search Helpers
SKills:
Popular SKills
Location (optional):
Search
Helper Search Results

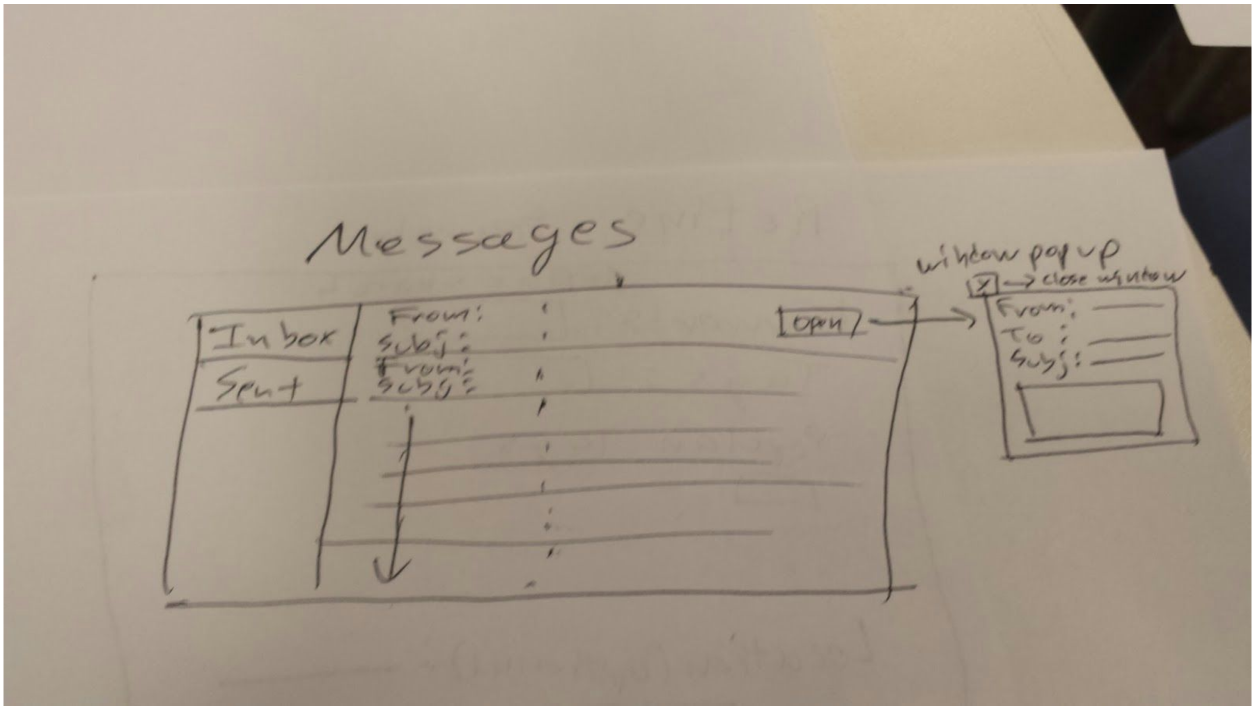

Helper Search Results
Search Helpers
Helper Results
Sort by:
user info
1) Username:
SKills :
~~Reviews:~~
Avg. Rating:
2)

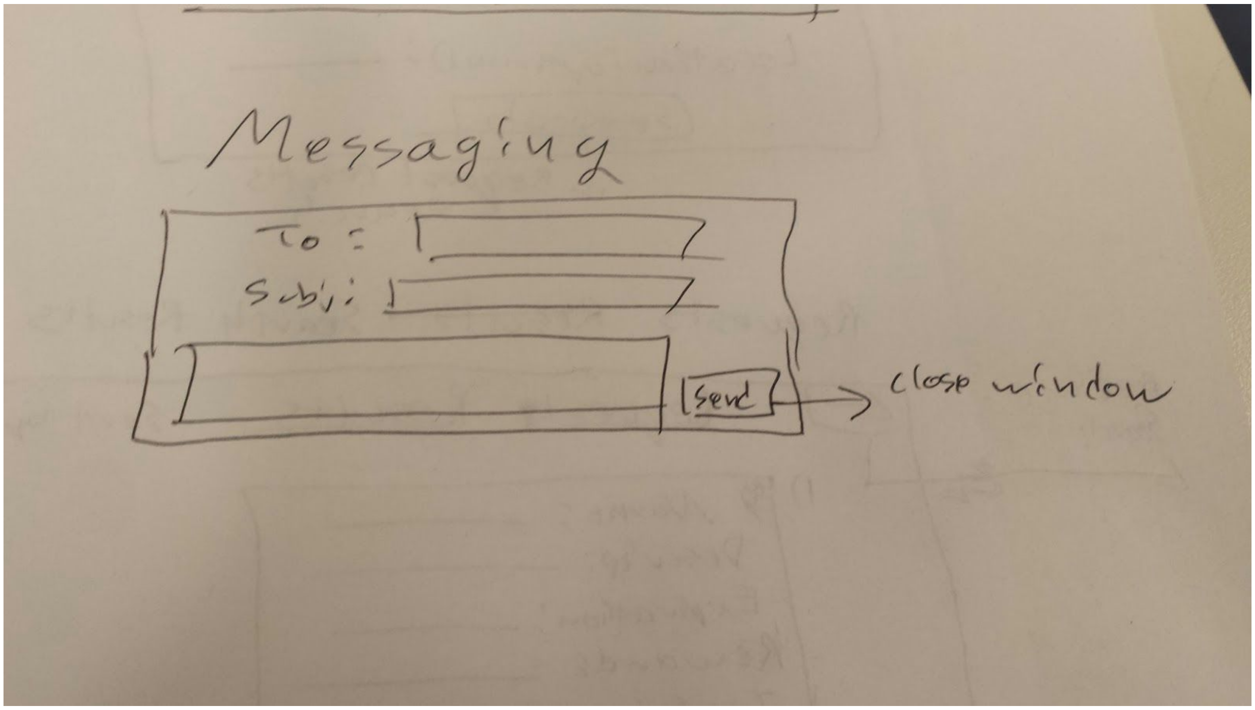


Refine Search



Requests ~~Results~~ Search Results


Messages
Inbox
Sent
From:
Subj:
From:
Subj:
Open
window pop up
X
close window
From:
To:
Subj:


Messaging
To:
Subj:
Send
close window

Design Challenges

Tags

- Option 1: Predefined Tags. This would allow us more explicit control and a better user experience for the exact tags we set, but runs the risk of being completely inadequate for the requests users actually make (i.e. not having a “Moving” or similar tag for someone who’s moving).
- Option 2: User-defined Tags. This provides more flexibility in categorization, which helps users feel more in control of how requests are categorized, but allows for different-yet-similar tags like “html” and “html5” to exist.

We decided on user-defined tags, as the existence of different-yet-similar tags can be resolved with something like a list of all tags or autocomplete in a tag search.

User Interaction

- Option 1: Uber for things. We could have the application such that helpers put up their availability and what kind of tasks they are willing to do and then when someone makes a request they can just choose from any of the helpers that meet the requirements. This would force helpers to set aside blocks of time where they should be ready to be called on at any time similar to Uber drivers.
- Option 2: Taskrabbit for MIT. We model the application with requesters posting their request so that other users can look through requests and easily choose the ones they want to do that are more convenient for them. This allows for more flexibility on both sides, but also makes it so it might be less likely for your request to be completed as there are no set of “helpers” that are guaranteed to be around to help.

We ended up deciding on the second idea since we do not want the site to become a job where helpers just sign up and wait to get money for menial tasks. Helpers instead look for ways to help the community around them when they have the time.

Rewards

- Option 1: Money only. To more strictly enforce rewards, one option is to restrict rewards to only being monetary and use something like the Venmo API.
- Option 2: Leverage the review system. Simply allow helpers to review requesters and encourage them to tell whether or not they got the stated reward.

We didn’t like the first option much because we want the app to be more focused on students in the MIT community helping each other out rather than monetary gain. We decided to go with the second option.