GrubSplit

Initial Design

By: Amanda Zhou, Marcos Pertierra, Matthew Miklasevich, Jorrie Brettin

Motivation

Ordering in groups is currently a cumbersome process riddled with problems. Here's how a typical group order may take place. First, one person decides they'd like to order food, but not enough to meet any sort of delivery minimum. That person - we'll call them the creator - needs to get a group together. They go around their living space knocking on doors and asking if anyone would be interested in ordering. Eventually, the creator has gotten a group of others - we'll call them joiners - together. None of the joiners have ever ordered from the restaurant the creator has chosen, so the creator's computer must be passed around so each joiner can consult the menu. By the time all the joiners have put in their individual orders, the entire order is a jumbled mess with no way to identify who has purchased what. In response, the creator must verbally confirm which joiner has ordered what, figure out how much that joiner owes, then charge them (via Venmo, Square, PayPal, cash, etc.). With no easy way to keep track of who has and has not paid, mistakes can be made which are ultimately costly for the creator. Once the food arrives - upwards of an hour later - the process of rounding up the joiners must be done again. Furthermore, without a record of who ordered what, joiners may end up getting the wrong food, leading to disgruntled joiners and a botched group ordering experience for all.

GrubSplit is designed to streamline ordering in groups by allowing for parallel processing by multiple users as well as automating most of the process. GrubSplit will allow users to gather groups en masse with a single touch, eliminating the need for gathering people and convincing them to join your order. The food selection process can be done by all users simultaneously instead of each needing to look at a single menu. Calculating cost and charging will be an automated task instead of careful bookkeeping. Notifying everyone when the food arrives will be just one more button press away. Finally, ensuring that everyone pays and gets the right food will be simple - the creator of the order will have a checklist with the users' names, orders, and payment status. GrubSplit will save time, money, and hassle for everyone involved in a group order.

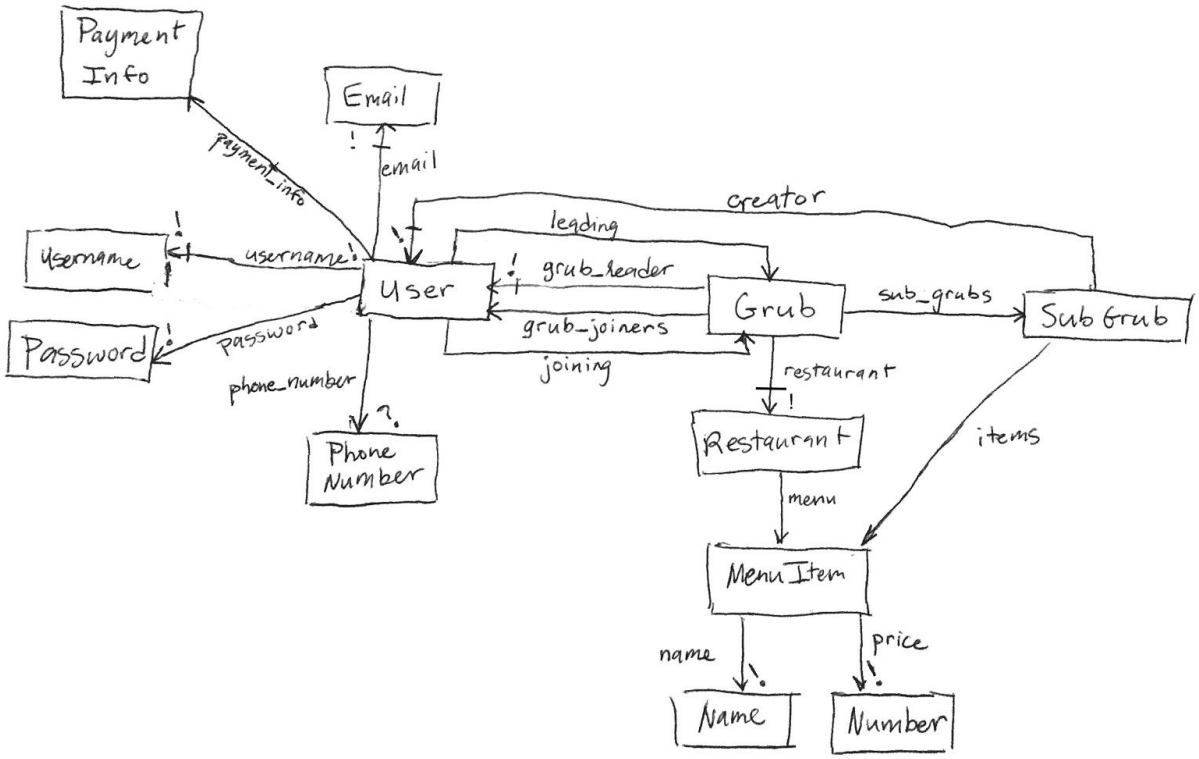
Concepts

GrubSplit is centered around a few core concepts which vastly simplify the process of ordering food in groups. There are two general categories which these concepts fit into: a user and an order.

First is the notion of a user. A GrubSplit user has a name and can initialize or join orders. A user who initializes an order is called a Creator. Creators have administrative rights over an order, which means it is their responsibility to decide from where the order is placed, when it is scheduled to arrive, and how it shall be dispersed. They also have responsibility as the person who pays the restaurant for the food in full, as well as who gets the food to others on the order. Those others can be called Joiners. Joiners don't have any administrative input; they are simply joining an already existing order from a specified restaurant at a specified time. Their responsibility is to submit their order in time and to pay the Creator upon fulfillment of the order. A vital part of GrubSplit is categorizing the users into groups. Doing so allows for the self selection of users into convenient classifications such that they can efficiently find others to join their order by inviting an entire group at once. For example, a user who is part of her sorority's group can invite all her sisters to join in an order with a single button press.

Next is the notion of an order. To clarify things, there are two distinct notion of what makes up an order. First, the order itself is called a Grub. A Grub, which is initialized by a Creator, is essentially the framework for a traditional food order, but without the food. That is, a Grub is from a specific restaurant and will be paid for in full by the Creator. The food items come from what are called SubGrubs. Any user participating in a Grub creates a SubGrub, which is a selection of food from the restaurant associated with the Grub. Each SubGrub submitted by a user has an associated username, food list, and cost. These properties will allow for the automated tracking of who got what and how much they owe. SubGrubs eliminate the need for the Creator to manually calculate and charge individual users or to ensure the right people get the right food.

Data Model



Textual Constraints

The creator of a SubGrub must either be a GrubLeader or GrubJoiner of the Grub that the SubGrub belongs to

Each username must be unique

A SubGrub's items must belong to the same restaurant of its Grub

The GrubLeader of a particular Grub must be the same user that is leading the Grub

The GrubJoiner of a particular Grub must be the same user that is joining the Grub

Design Insights

Each Grub is associated with a single restaurant. If the GrubLeader wants to change the restaurant, he/she must cancel the Grub and create a new one.

In the current data model, menu items could be shared between different grubs. In practice, this means that different orders can contain the same item. This isn't an issue because each menu item refers to the food's name and price, not the food itself.

Security Concerns

Overview

Each GrubSplit user's profile will contain a username, a password, and payment information. Payment information is incredibly sensitive and any exploits that can access user's bank accounts could be devastating, so security is absolutely critical.

Threat Model

We anticipate that malicious hackers will want to exploit GrubSplit for a number of reasons. If a user's account is compromised and logged into by an attacker, he could order food from restaurants and the unsuspecting user will end up paying for these meals. In the worst case scenario, an attacker could conceivably gain control of a user's payment information and steal money from their bank account.

We expect unauthenticated users to be able to construct requests in order to gain access to accounts or otherwise tamper with orders.

Mitigating Standard Attacks

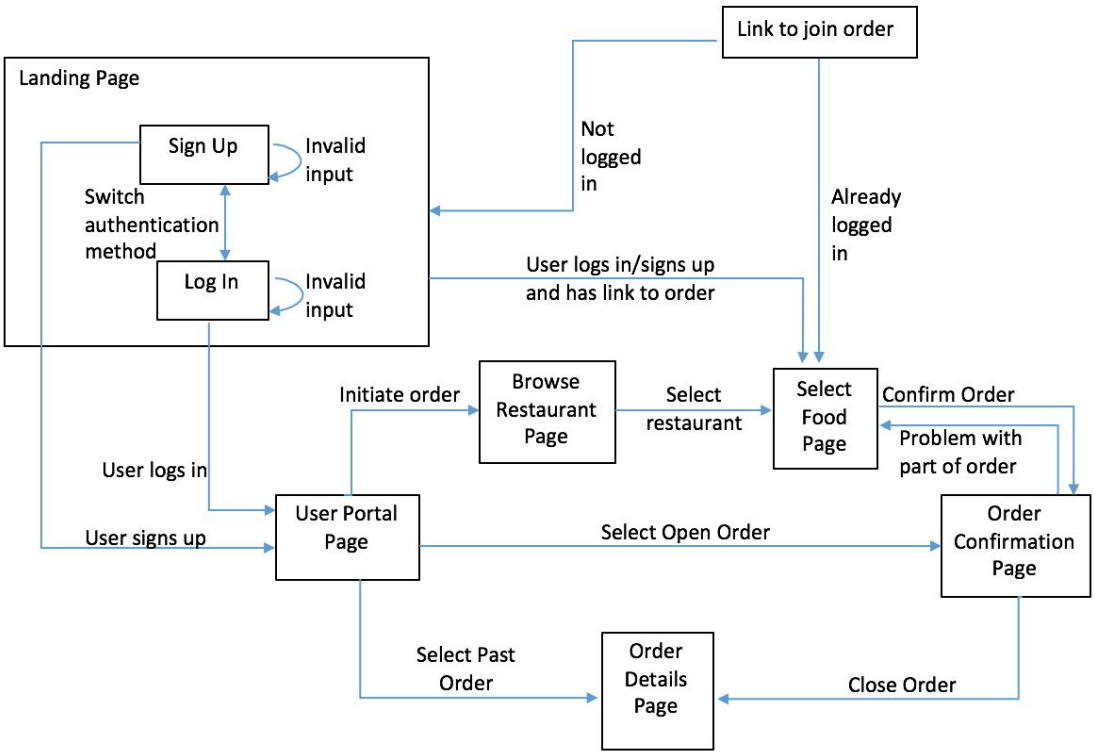
Passwords will be stored in a database, but GrubSplit will never store a user's plaintext password. Instead, we will salt the password and hash it using the bcrypt API. This will prevent attackers from gaining access to user's accounts by intercepting packets containing login information sent over the network. Furthermore, salting passwords prevent rainbow table attacks, where hackers build up a table of passwords and hash values then guess . Finally, we will require GrubSplit to be server over HTTPS rather than HTTP to ensure that all traffic is encrypted.

We aim to prevent XSS (cross-site scripting) by properly escaping all points of user-input. GrubSplit will only require user input through text fields when users register, login, submit payment information, invite friends to an order, and search for restaurants. This will prevent malicious users from running arbitrary scripts in GrubSplit to exploit vulnerabilities and gain access to otherwise secret data. It will also prevent hackers from entering database queries into text fields and gaining access to, modifying, or deleting someone else's data.

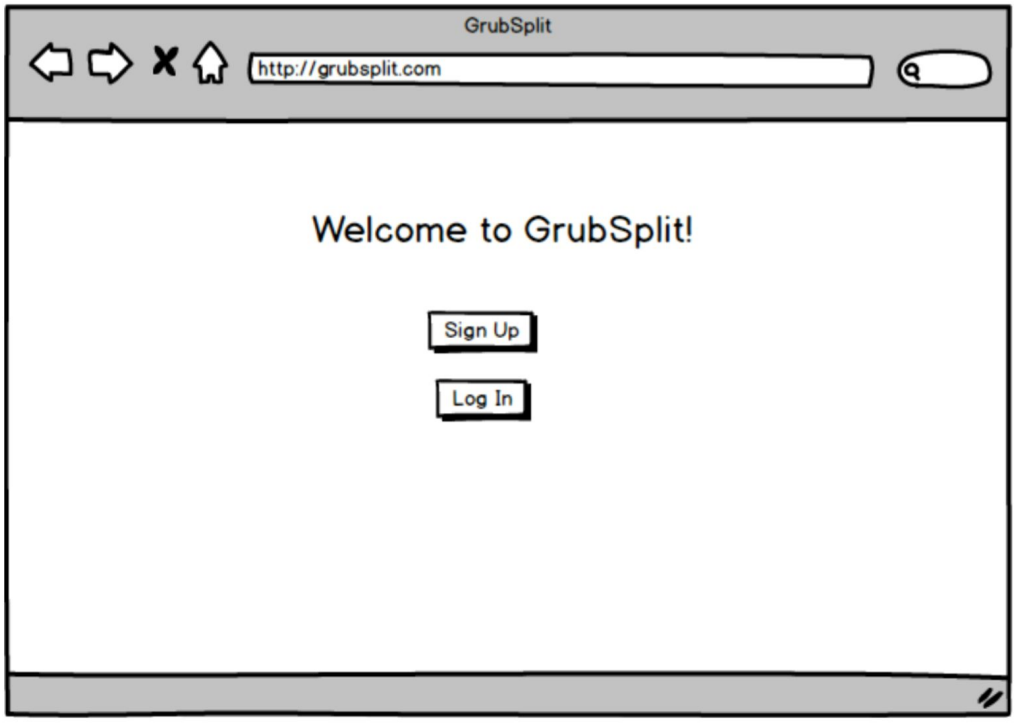
We aim to prevent CSRF (cross-site request forgery) by using hidden form tokens that are session-specific. This ensures that the request was sent by the correct, expected client. The security of this solution is enhanced by using HTTPS to encrypt all client-server traffic.

User Interface

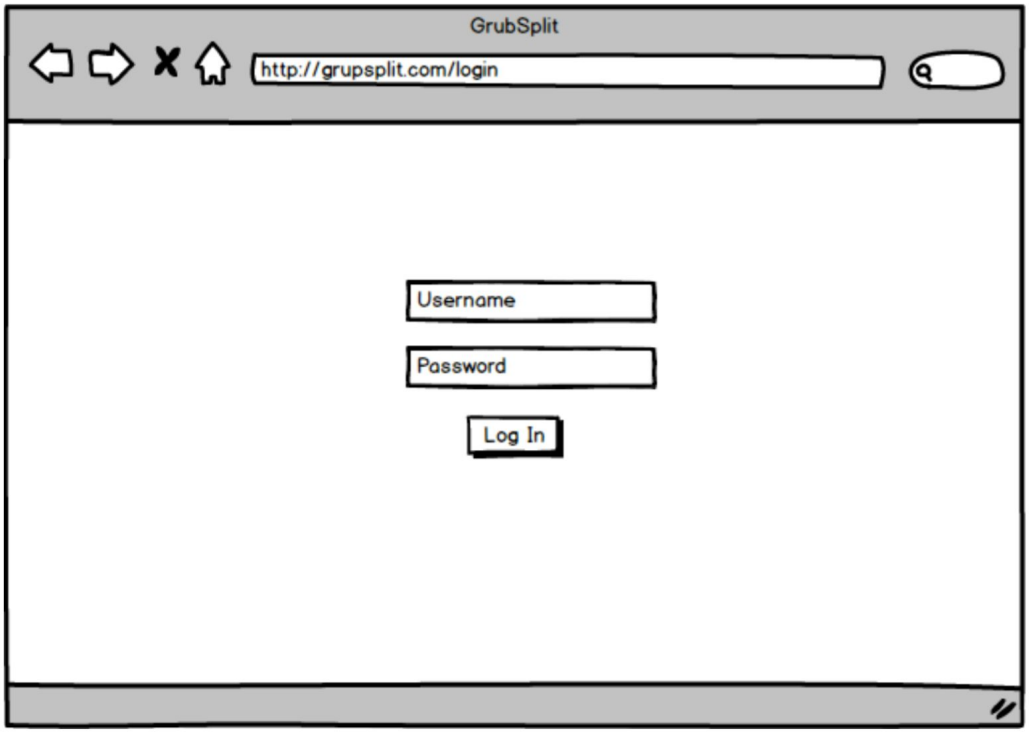
Transition Diagram



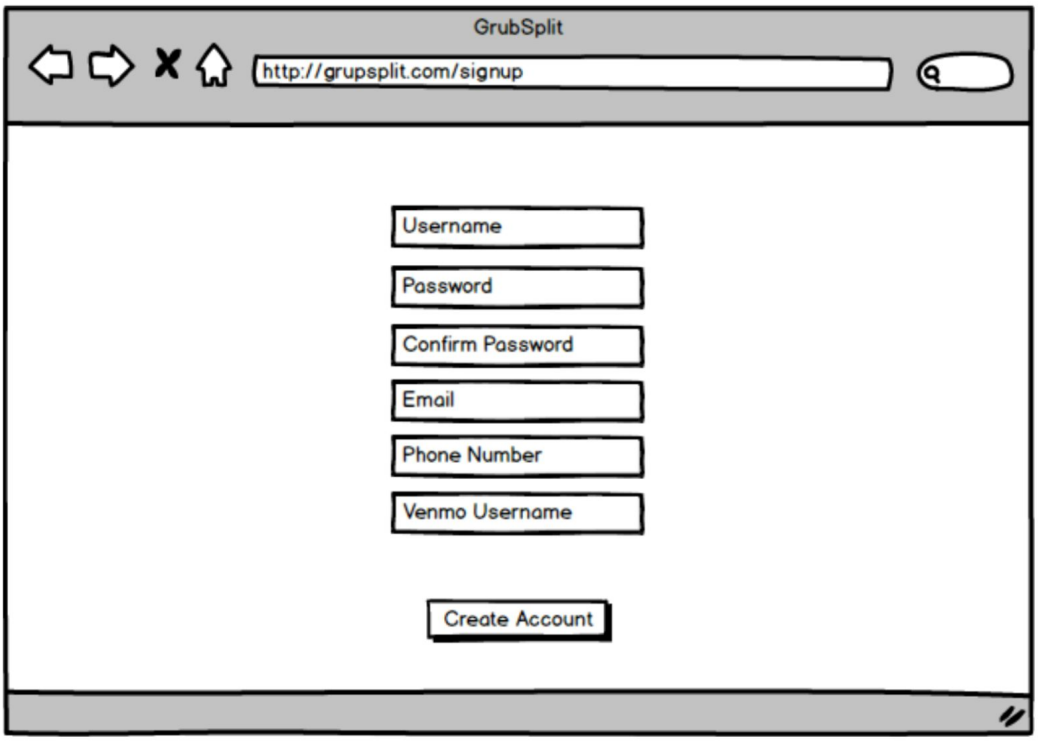
User Landing Page



Login Page

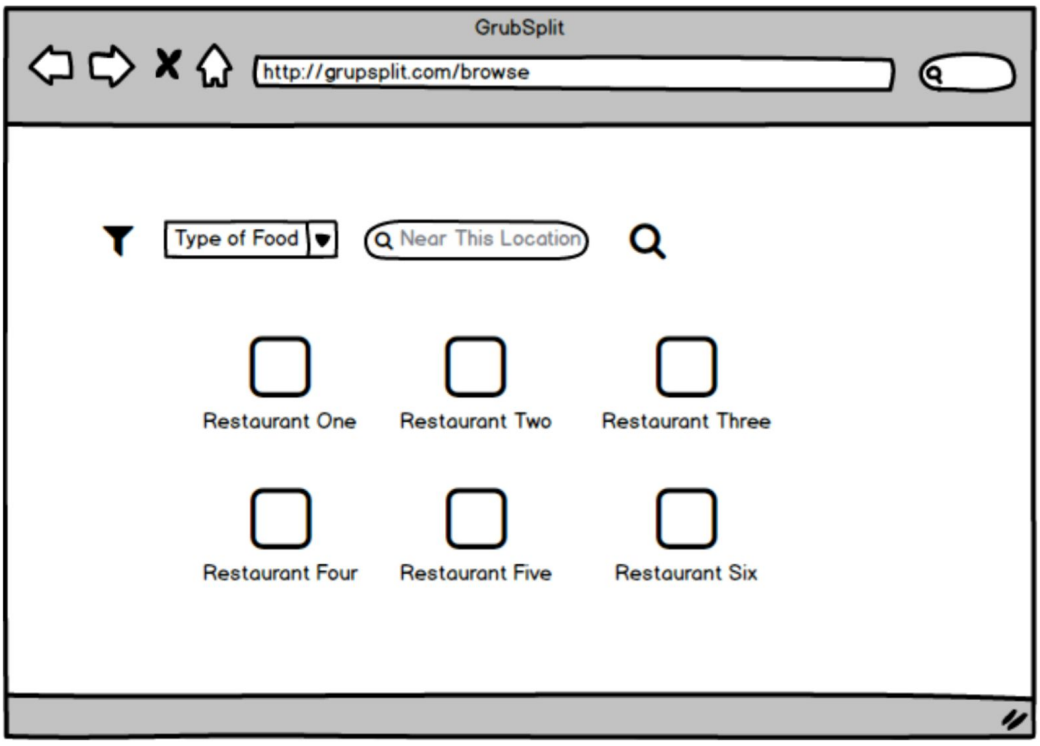


Sign Up Page

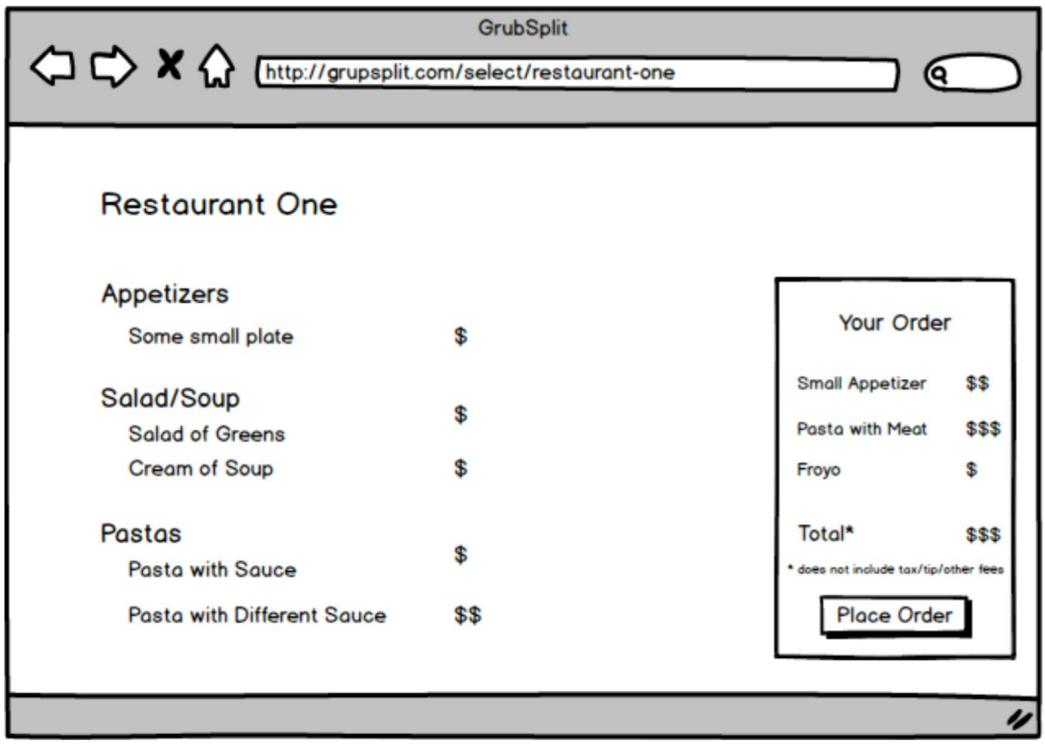


User Portal Page

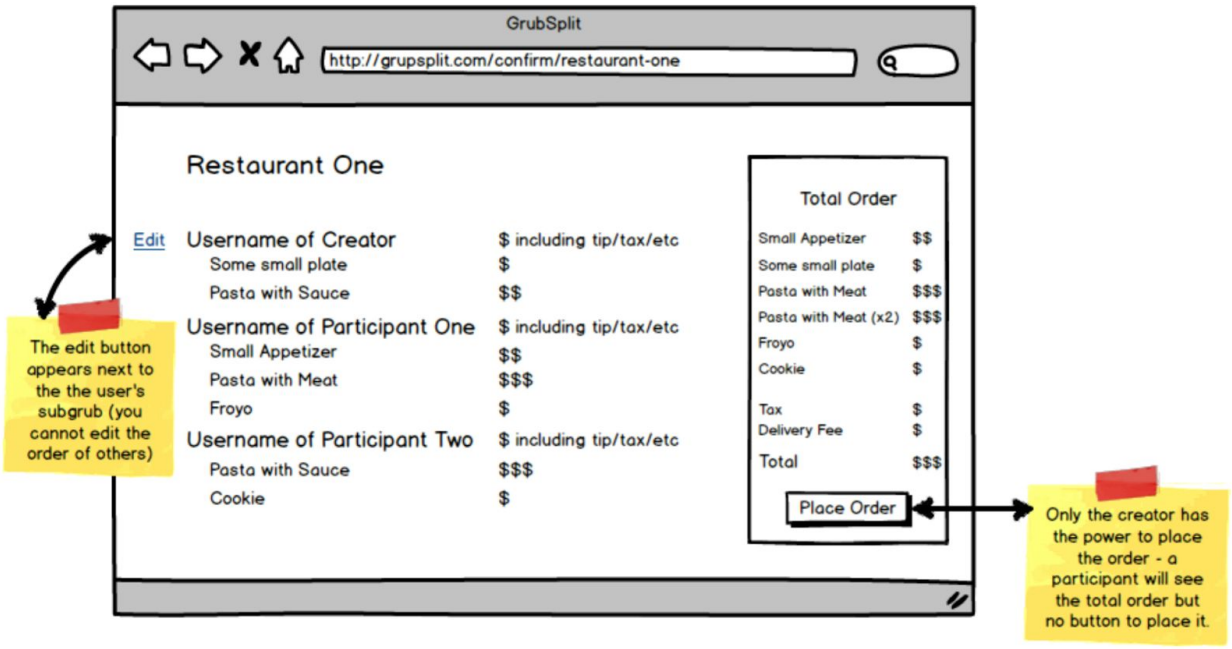
Browse Restaurant Page



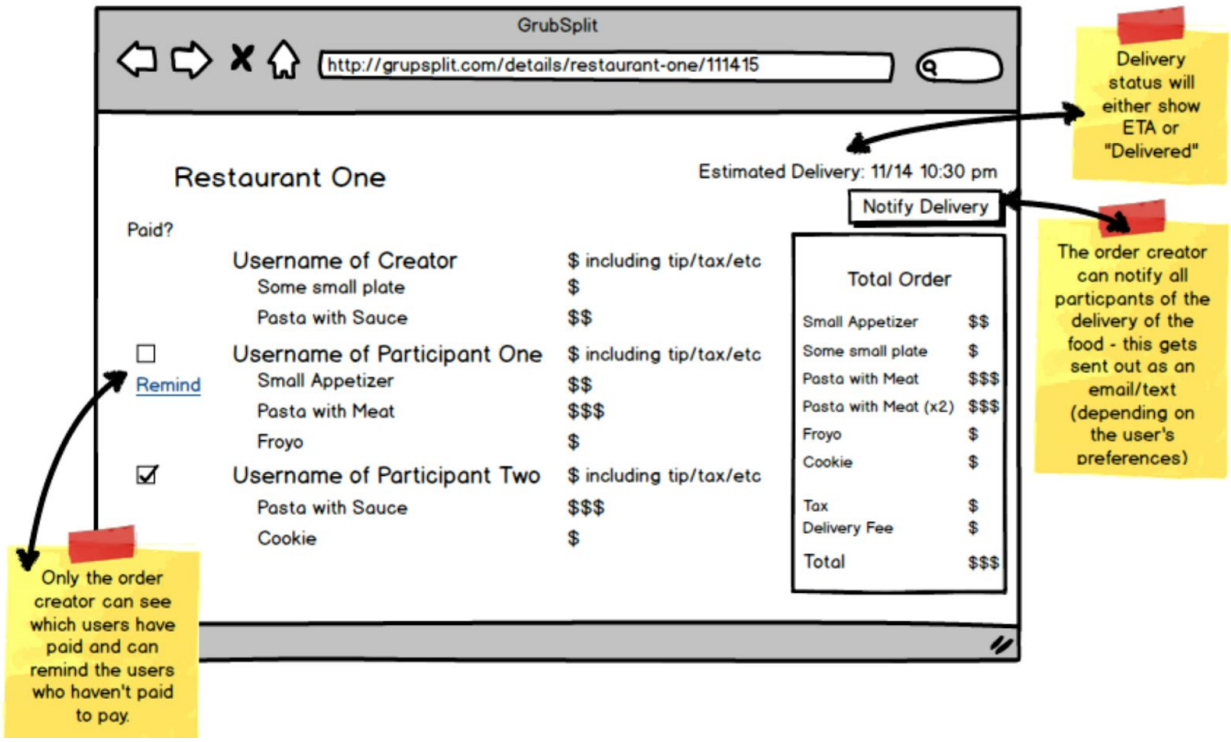
Select Food Page



Order Confirmation Page



Order Details Page



Challenges

Problems

Relationship between Users, Grubs, and SubGrubs in data model

It was difficult deciding how exactly to map the relationship between a user, their individual SubGrubs, and the complete Grubs themselves. We decided to directly map users to their Grubs, with the relationship “joining” or “leading”. Then, there is a relationship between Grub and SubGrub where a Grub consists of multiple SubGrubs. Finally, a relation maps each SubGrub to its owner. Although it was tricky to describe and capture this behavior at first, clarifying this aspect of our data model helped us design and soon implement a more succinct, efficient user experience and UI.

We considered directly mapping the “joining” and “leading” relationships to SubGrubs, but it would have been difficult to determine or enforce who the single GrubLeader of the Grub actually is. With our current data model, it’s easy to display a list of orders that a user is a part of, and tell whether or not he is leading those orders. The Grub can easily show the included SubGrubs, as well as their owners.

Distinguishing order creators and joiners

From a conceptual standpoint, there is a clear difference between Creators and Joiners. However, when it came to the user interface, we had the decision of creating two order pages (one for Creators and one for Joiners) or just one. Since the Creators have “admin” access to the Grub, they should see all information regarding the SubGrubs. However, for the Joiners, the question was: should they only be able to see their SubGrub or be able to see the entire Grub? We chose to allow all users to see details of the entire Grub. This ensured continuity between the user experience, regardless of being a Joiner or Creator. Additionally, we saw no real benefit to showing users less information regarding the entire Grub. Displaying the entire Grub makes it explicit how the final cost was calculated (by seeing what others ordered, it is more clear why you were charged a certain amount of tax/tip/delivery fee) and may even help people decide what food to order in the future.