Yootles: A Social IOU System

Y$Id: yootles.texi 539 2008-04-22 09:07:00Z dreeves $

**Daniel Reeves, Tejaswi Kasturi, David Pennock,
Sandeep Rathour, Prasenjit Sarkar,
Bethany Soule, George Levchenko**

Yahoo! Research

Table of Contents

1 The Yootles System

Yootles is a system for declaring IOUs between people on a social network. It is designed to allow people to jot down who paid for dinner or other transactions that people often want to but forget to keep track of. But it's useful for much more. It can be used for things like tracking rent payments, sharing utility bills, even tracking interest on personal loans. And it supports non-monetary currencies. Yootles in fact refers to the seminal currency in the system: an abstract measure of happiness or utility used in a suite of forthcoming applications including voting and wagering and various incentive mechanisms.

The next incarnation of the Yootles IOU system will support arbitrary IOU routing (see http://yootles.com/trustnets.pdf). The current version supports bilateral accounting of IOUs with some advanced accounting features. It will always be possible to export your transaction history in an open format so you will never be tied to any particular implementation. (You wouldn't use Flickr without assurance that you could always retrieve your photos—Yootles treats your IOUs the same way.) In fact, your IOUs are always stored exactly as you type them so you can always independently verify everything the Yootles system tells you about your accounts and your balances.

1.1 Yootles Accounts vs. Yootles Users

A user is a person who logs in to the Yootles system. An account is an entity that can issue and receive IOUs. A user may have several accounts, perhaps subaccounts to keep track of different spending categories, or accounts for friends who are not actual users in the system. Accounts may also be created for purposes such as charity fundraising that aren't tied to any particular user. All accounts are specified by an account group and an account name within that group, separated by a colon (e.g., `"smithfamily:alice"`). New users creating their first accounts need not know about the concept of groups—in that case a default account is created by making the group name the username and the account name the user's initial(s) or first name, e.g., `"alice:ac"` or `"alice:alice"`. Note that, while group names must be globally unique, account names must only be unique within groups. For example, `jets:bob` and `sharks:bob` are entirely distinct accounts. By default, all users have access to all accounts and simply issuing an IOU to or from a nonexistent account creates that account. We describe this open access philosophy and how to limit it, along with other relationships between users and accounts below (see Section 1.8 [Access Control], page 4).

1.2 IOUs

At its most basic an IOU is a simple transfer from one account to another. We call this an *atomic IOU*. Yootles also supports two generalizations: repeating IOUs and multilateral IOUs. Fundamentally, all IOUs are broken down into atomic IOUs, but Yootles does not store them this way[1] because this would clutter the transaction history and in fact there is additional information implicit in the fact that an IOU came from a repeating or multilateral IOU that would be lost if all IOUs were atomized.

From a user's point of view a *settlement* is the opposite of an IOU. The Yootles system, however, has no conception of settlements. When someone makes good on (settles) a past

[1] This is not literally true. It stores atomic IOUs but only for caching/computational purposes.

IOU it is recorded as simply another IOU in the opposite direction. We leave it to the user interface to add whatever layer is deemed appropriate to make that intuitive. (Note that this invariably trips people up.)

1.3 Repeating IOUs

A repeating IOU specifies a time interval that the IOU automatically repeats, e.g., "every 2 weeks". IOUs can repeat indefinitely or end at a specified time. When an end time is specified it is not taken to be the time of the last IOU but rather a time after which no further IOUs will be generated. The last repeated IOU is prorated for the amount of time between the last IOU and the end time, compared to the repeat interval. For example, a Y$60 IOU that repeats semiannually, issued on 2008 January 1 and ending 2009 April 1 will be atomized into:

- Y$60 on 2008 Jan 1
- Y$60 on 2008 Jul 1 (6 months later)
- Y$30 on 2009 Jan 1 (another 6 months later, but with only 3 months till IOUs stop)

Note that, consistent with the above interpretation of end date, if the end date coincides with the last IOU date then the last payment will be prorated to zero. The reason for this interpretation of the end date is that it preserves the property that the total amount paid divided by the elapsed time between start and end is always exactly equal to the IOU amount. User interfaces may want to highlight the prorated amount of the last payment to avoid any confusion. The API makes it easy to do that.[2]

1.4 Multilateral IOUs

Multilateral IOUs are IOUs that are shared between some set of accounts. Instead of specifying a single issuer account and recipient account, either or both of these may be a set of accounts, specified as mathematical expressions in which the accounts are treated as variables and the corresponding coefficients indicate proportions by which the IOU amount is split between the issuers or recipients. This turns out to be a simple and elegant way to specify group IOUs, as the following examples illustrate:

Multilateral IOU	**Atomic IOUs**
10 from `alice` to `bob+carol`	• 5 from `alice` to `bob` • 5 from `alice` to `carol`

Multilateral IOU	**Atomic IOUs**

[2] Admittedly, this interpretation of the end date will be confusing to users but the alternatives are worse. For example, having one repeating IOU pick up where another ends (perhaps the rent went up) will work correctly under this model.

30 from `alice+2bob` to `carol`	• 30*1/3 = 10 from `alice` to `carol` • 30*2/3 = 20 from `bob` to `carol`

I.e., Alice nets -10, Bob nets -20, and Carol nets +30.

Multilateral IOU	**Atomic IOU**
20 from `alice+bob` to `carol+deb`	• 10 from `alice` to `carol+deb` • 10 from `bob` to `carol+deb` **which becomes** • 5 from `alice` to `carol` • 5 from `alice` to `deb` • 5 from `bob` to `carol` • 5 from `bob` to `deb`

I.e., Alice nets -10, Bob nets -10, Carol nets +10, and Deb nets +10.

In the following example, Alice and Bob have dinner and Alice orders a $7 dish, Bob a $9 dish, and the bill with tax/tip comes to $20. They each chip in $10.

Multilateral IOU	**Atomic IOUs**
20 from 7`alice+9bob` to 10`alice+10bob`	• 8.75 from `alice` to `alice+bob`[3] • 11.25 from `bob` to `alice+bob` **which becomes** • 4.375 from `alice` to `alice` (this is a no-op) • 4.375 from `alice` to `bob` • 5.625 from `bob` to `alice` • 5.625 from `bob` to `bob` (also a no-op)

The above has a net effect of 1.25 from Bob to Alice. This is the precise amount, allocating tax/tip proportional to meal costs, that Bob owes Alice after she put in more than her share—10 instead of 7/16*20—for the meal).

The last example shows how—once you're used to the powerful and perhaps initially intimidating language for expressing them—you can track complicated multilateral IOUs by specifying very concisely the nature of the events that occasioned them. This will turn

[3] Note that `"10alice+10bob"` is equivalent to `"alice+bob"`, i.e., they each receive half, or $10 out of $20.

out to be valuable for various Yootles decision and prediction mechanisms. Interfaces can sugar-coat this underlying language in various ways for less sophisticated users. In the initial release of the Yootles Facebook application, the full power of multilateral IOUs will not be exposed at all.

1.5 Currencies

Currencies can be created for anything—hours of babysitting, borrowed books, beers, ounces of gold, points, kudos, and even, of course, units of happiness. All currencies are publicly available to everyone. There is no such thing as a private currency, just obscure ones. The Yootles system does not track exchange rates between any currencies. It tracks your balances for each currency you have dealt with and keeps them entirely separate.

1.6 Interest

Every pair of accounts at every moment in time has an interest rate (which applies to all currencies) that causes positive balances to become more positive and negative balances to become more negative (or vice versa if the interest rate is negative, which is allowed[4]). Anyone who can issue IOUs for both accounts (see the Access Control section below) can change the interest rate for an account pair at any time. Interest rates can also be changed retroactively.

1.7 Credit

Every directed pair of accounts has a credit limit for each currency, initially zero. For example, Alice can state that she trusts Bob up to 10 yootles, meaning that she is willing to issue her own IOU of 10 yootles in exchange for receiving an IOU from Bob for the same amount, assuming the interest rate differential is not unfavorable. Credit limits are the basis of automatic IOU routing (see http://yootles.com/trustnets.pdf). Credit limits can be specified in the current Yootles system but automatic IOU routing is not done.[5]

1.8 Access Control

As explained above, there is a strict conceptual separation between Yootles accounts and users. All of the back-end processing is done strictly in terms of accounts, and users can manipulate those accounts according to an access policy. Specifically, there are six flags that specify a given user's relationship to a given account (we use a running example with `alice` as a representative user accessing the account `jets:bob`):

[4] And conceivably desired if you wanted to formalize the informal social practice of gradually forgiving debts over time.

[5] In a future release we will consider whether it should be possible for different chunks of credit to have different interest rates. We decided against the extreme of having every IOU be able to specify a different interest rate because people can always approximate that by using separate subaccounts. A fully general specification of how much I trust some user Alice means being able to specify the interest I would require as a function of my balance with her. For example, when my balance is negative (I owe Alice money) then any interest rate, positive or negative, is acceptable (I'll just pay up if it's too high), 0% may be acceptable for balances up to +20 (i.e., Alice owing me $20), 5% up to $200, 10% up to $1000, and infinity% beyond that which is equivalent to setting a hard limit of $1000. But even that level of generality still misses something, namely liquidity—the interest rate should depend on how long a balance is outstanding. So until we get this aspect of the design fully baked we'll let it suffice to give users full manual control over their interest rates.

- `root` – Having root access on an account means being able to change access settings for other users. If user `alice` has root access on `jets:bob` then `alice` was the original creator of `jets:bob` or had root status conferred by another root user. By definition, only root users can change this flag, with one exception: if no user has root access to an account (i.e., the sole root user revoked their own root access) then the account is considered an orphan and *any* user can set root access. (It is thus safer to transfer root access by setting it for the new user before revoking it for the old user.)
- `view` – Having view access on an account means being able to view IOUs involving that account. For example, if `view` is true for `<alice, jets:bob>` then `alice` can view all IOUs involving `jets:bob`. Only root users can change this flag.
- `ctrl` – Having control access on an account means being able to issue IOUs from that account. With `ctrl` access, `alice` can issue IOUs from `jets:bob`. Note that this is the default for all users and all accounts. Only root users can change this flag.
- `main` – When `main` is set to true for a user/account pair, it indicates that that account is the user's main account. Designating account `jets:bob` as `alice`'s main account means it is the one referred to by the special account designating syntax `"[alice]"`. I.e., square brackets around a username is syntactic sugar for the main account of that user. Anyone with `view` and `ctrl` access can make an account their main account unless it is already the main account of someone else. A user need not have a main account (and won't have one until they choose or create one) in which case using the square bracket designation for that user will be treated as an error.
- `mine` – This field is typically treated as boolean but can in fact be set to any real number in [0,1] where 0 (`"not mine"`) is interpreted as false and 1 (`"all mine"`) as true. For a user to have a `mine` setting of x for an account means that a fraction x of that account's balance should be included in the user's net balance. Conceptually, this field indicates what fraction of an account belongs to the user. Under this interpretation, the `mine` settings for all users for a given account should sum to one, though this is not enforced. As an example, if `mine`=1 for `<alice, jets:bob>` then account `jets:bob` is treated as `alice`'s subaccount and is included in `alice`'s net balance. Anyone with `view` and `ctrl` access can make an account their subaccount (to any degree).
- `ntfy` – Having `"notify access"` on an account means you receive notifications when the account receives or issues an IOU. (E.g., `alice` would be notified whenever `jets:bob` receives or issues an IOU.) Only root users can change this flag, except that anyone can turn off notifications for themselves, i.e., unsubscribe.

The following constraints should hold among these flags:

- `main` ⇒ `view` & `ctrl` (you should always be able to view and control your main account)
- `mine` > 0 ⇒ `view` & `ctrl` (you should always have control over your subaccounts)
- `main` ⇒ `mine` = 1 (your main account should always show up in your net balance)

Note that `ctrl` does not imply `view`. (The default is `ctrl` and `view` access but it is possible to have only `ctrl` access but not `view` access to an account.) Also note that `mine` does not imply `root` since anyone can claim an account as their main account or subaccount. This means you can do most of what you might want with no permission but will still have to ask for permission to change the access of others. Notification is critical to such a liberal access policy. Evil Eve can issue IOUs to herself from random accounts all day long but users who manage those accounts can quickly repudiate them.

Finally, consider the case of the user `alice` creating the account `jets:bob` for Bob who is not currently a user. No setup is required on Alice's part—issuing an IOU to or from `jets:bob` automatically creates that account. But instead of or in addition to specifying an account name, a front-end using the Yootles API may allow Alice to specify Bob's email address. In that case the system will create a new user and a new account for Bob and invite Bob to claim the account. Bob can change his username and account name if he wants[6] but if he only has one account the interface should keep him blissfully ignorant of the concept of multiple accounts at all.) The following flags will be set for Alice and Bob and the new account (which we'll continue to refer to as `jets:bob` for simplicity):

- user `alice`, account `jets:bob`
  - `root` = yes (since Alice created the account)
  - `view` = yes
  - `ctrl` = yes (this is the default for all users and accounts)
  - `main` = no
  - `mine` = 0
  - `ntfy` = yes
- user `bob` (though Bob does not yet have a username), account `jets:bob`
  - `root` = yes (since Alice specified a new user for this account, we assume she means to confer root status)
  - `view` = yes
  - `ctrl` = yes
  - `main` = no (only Bob can set an account to be his main account)
  - `mine` = 1
  - `ntfy` = yes (this will be moot if Alice did not provide an email address for Bob)

Future versions of Yootles may support a richer language for blocking classes of users.

[6] Technically he can't change his account name, he can just transfer the balance and copy the interest rates and credit limits to a new account and ignore the old account, but this is encapsulated by the `merge` command (see Section 5.15 [merge], page 46).

2 Use Cases

2.1 Issuing IOUs between Facebook users

The Yootles Facebook application is currently the primary client for the Yootles API. Suppose Alice is a Facebook user who just installed the Yootles application. Alice would like to give Bob an IOU. The system should prompt Alice to choose a Yootles username (with an auto-generated default available based on her full name) and a nickname for herself, defaulting to her initials (possibly even a single initial). Suppose her username is alice and her nickname is alc. The system creates the username `alice` and creates the account `alice:alc` as Alice's main account. Note that, although the group `alice` is created in this process, Alice will never need to know about the concept of groups as long as she remains a casual user.

Next, Alice enters the amount and a short string giving the reason for the IOU and selects her friend Bob as the recipient via the usual "start typing a friend's name" interface. More complicated IOUs can be issued by setting other fields but everything except the amount, the recipient, and the reason have reasonable defaults (see Section 5.8 [owe], page 30 for details). If Bob has a main account already, say `bob:b`, the system uses it and Alice sees in her transaction history an entry like "123 from `alice:alc` to `bob:b` for 'lunch' on 2007-12-04" (possibly with the "alice:" group prefix redacted). Otherwise, Alice is prompted to choose a nickname for Bob, say "bob", which the system will use to construct an account for Bob using Alice's group, e.g., `alice:bob`. In this case there is no reason to show the group prefixes and Alice sees an entry like "123 from `alice` to `bob` for 'lunch' on 2007-12-04".

Bob will be notified that Alice has given him an IOU (as will Alice), with a link to edit it or delete it,[1] but the system requires no action from Bob. Carol might go through the same process as Alice, leaving Bob with two accounts, `alice:bob` and `carol:bob`. Both accounts will be marked as belonging to Bob (via the `mine` field in the access table) but neither will be marked as his main account unless he chooses one. Or he can create a new account altogether and consolidate the balances and close the accounts `alice:bob` and `alice:carol` (see Section 5.15 [merge], page 46).

2.2 Issuing an IOU in Facebook to someone not on Facebook

This case is similar to the previous one but instead of selecting Bob via the Facebook-style interface, Alice supplies an email address for Bob. The system checks if there is a Yootles user with that email address and, if so, whether that user has a main account. If so, the email address is all that is required. Otherwise, Alice chooses a nickname for Bob and the process continues as above with a new user created for Bob with whatever contact information, such as email address, that Alice provided. Alice can also use the same process without providing an email address or any contact information for Bob. In this case the account created for Bob amounts to merely a subaccount of Alice's. Bob may or may not end up taking control of it in the future.

The notifications Bob receives will be something like: "Alice has given you an IOU for ___. Click here to take control of your account or click here to stop receiving these

[1] That is, replace it with a zero IOU.

notifications." Nothing prevents Alice from issuing and IOU *from* Bob as well (equivalent to a negative IOU to Bob), in which case the notification will be: "Alice has marked you as owing her ___. Click here to take control of your account (or edit/repudiate this IOU) or click here to stop notifications."

2.3 Fund-raising with Yootles

Dan announces to his friends that he is fund-raising for the American Lung Association (ALA) and that if anyone would like to donate yootles just transfer them to the account `yooniversal:ala`.[2] Yootles have no exchange rate with real money but Dan pledges to his friends that however many yootles end up in the `ala` account, he will donate that many dollars to the ALA (and then transfer the yootles from `yooniversal:ala` to himself). In effect, his friends are promising him influence in decisions, potential bragging rights in wagers, and other forms of "yootility" in exchange for his donation to a worthy charity.

2.4 Shared water bill

Housemates can create a common group and easily do shared accounting within that group (as well as with people outside of it). Multilateral IOUs are useful for this. Suppose the housemates each have accounts: `elmstreet:alice`, `elmstreet:bob`, `elmstreet:carol`. A typical IOU might be entered as "$100 from '`alice+bob+3carol`' to `bob` for 'water bill that Bob paid for and for which Carol used 3 times as much water as anyone else' in the group `elmstreet`." The previous sections on multilateral IOUs and repeating IOUs (useful for rent payments!) have more details about group accounting (see Section 1.2 [IOUs], page 1).

Note that this kind of group accounting does not require buy-in from all the members. We have found that it is typically a minority of people in a group who are motivated to track IOUs but that almost everyone is happy to be included so long as they aren't relied upon to do anything. By default, accounts such as `elmstreet:carol` are open for anyone to view and issue IOUs to and from. All IOUs are repudiable so access can be tightened as it is deemed necessary.

2.5 Renaming an account

Account names, like `jets:bob`, are treated as unique identifiers in Yootles—there are no account numbers. This means the process of renaming an account is convoluted, but the interface can mask this. Here we describe the underlying process.

Suppose Bob has an account `old:bob` he wants to rename to or merge in with `new:bob`. If `old:bob`'s balances are all zero and Bob has root access he can revoke everyone else's access to `old:bob` and then simply abandon it. There is no way to actually delete it—this is because entries with `old:bob` may still exist in people's transaction histories. If `old:bob` has one or more nonzero balances, renaming consists of transferring those balances to `new:bob`. Say Alice owes `old:bob` 8 yootles. By issuing an IOU from `old:bob` to Alice for 8 yootles we zero the Alice balance Then we issue an IOU of 8 yootles from Alice to `new:bob`. Now

[2] This is based on a true story. We raised $1000 of real money for the Multiple Sclerosis Society using our initial prototype this way.

Alice owes `new:bob` instead of `old:bob`. We repeat this until all `old:bob`'s balances are zero, and then (optionally) revoke access to `old:bob`.[3]

The above process can't be entirely hidden from the end users. At the least, users who have balances with `old:bob` will see two new IOUs, one zeroing out with `old:bob` and one creating the balance with `new:bob`. This serves as a natural way to inform all those concerned about account name changes. But from Bob's perspective, the above process can be automated and the API provides a command (see Section 5.15 [merge], page 46) encapsulating the above steps of zeroing out balances in an old account and transferring them to a new account (note that nothing prevents the new account from already existing and having other balances).

[3] With automatic payment routing, Bob could also just issue an IOU from `old:bob` to `new:bob` and sever all credit limits coming into and out of `old:bob` except for a credit limit between `old:bob` and `new:bob`. Then the system will automatically reroute IOUs to go through `new:bob` and `old:bob` will be at zero with every account but `new:bob`. Bob then does one transfer between `old:bob` and `new:bob` to zero out `old:bob` completely.

3 Data Structures for Yootles

The fundamental data structures for accounts, account groups, IOUs, interest, credit, and currencies are given below. These are followed by data structures for users (people who can log in to the Yootles system), aliases (e.g., Facebook ID or email address) and access control (who can do what with which accounts). This is a subset of the underlying database schema. Data types are given in parentheses. Timestamps are integers specifying the number of seconds since 1970-01-01 00:00:00 GMT, i.e., unixtime. In the database, pointers are realized as integer IDs.

3.1 Data Structures Related to Accounts and IOUs

`group`

An account group name is prefixed to an account name, creating a namespace for a related set of accounts. The fields for the `group` data structure are group name and group description:

- `name` (string). Name of the account group, e.g., "jets" or "reevesfamily".
- `desc` (string). Text field describing the group.

`account`

An account (not to be confused with a user) represents an entity that can issue and receive IOUs, extend or accept credit, etc.

- `grp` (pointer to `group`). The group this account belongs to.
- `name` (string). The account name, e.g., "alice", "bob", or "mom". Note that the combination of `grp` and `name` must be globally unique and cannot be changed.
- `desc` (string). Text field describing this account.

`currency`

Currencies can include yootles, real money, time, even non-commodity items such as books (if you wanted to keep track of books you lent to friends).

- `code` (string). Symbol for the currency. Initial values: `{`ytl, usd, inr, can, beer`}`.
- `name` (string). Name of the currency. Initial values: `{`Yootles, US Dollars, Indian Rupees, Canadian Dollars, Beers`}`.
- `desc` (string). Text field describing the currency.

`rawIOU`

Raw IOUs, which may encompass multilateral IOUs and repeating IOUs, are stored in a way that corresponds directly to what prompted issuing the IOU (see Section 5.8 [owe], page 30).

- `amt` (string). The amount of the IOU, stored as a mathematical expression matching `/[\d\.\+\-\/\*\ \(\)]+/` and evaluating to a number.
- `from` (string).[1] The account(s) issuing the IOU, stored as a mathematical expression where the symbols are account names.

[1] This field is called `issuers` in the database as `from` is a mySQL reserved word.

- `to` (string).[2] The account(s) receiving the IOU, same format as `from`. The accounts in `from` and `to` can be prefixed with the account name (e.g., "yooniversal:alice") but if not, use the group specified by `grp`.
- `when` (timestamp).[3] Date/time of the IOU (need not be when it was actually issued).[4]
- `why` (string). Text field giving a short reason for the IOU.
- `rpt` (real). How often the IOU should automatically repeat (the period, not the frequency). Default: -1, meaning don't repeat.
- `rptunit` ({day, week, month, year}). The unit of measure for `rpt`. E.g., `rpt`=1/2 and `rptunit`=year means repeat twice a year (semiannually).
- `til` (timestamp). Time after which no auto-repeats (if the last repeated IOU falls before this time then prorate the last IOU). Default: -1, meaning forever.
- `cur` (pointer to `currency`). Currency. Defaults to yootles.[5]
- `grp` (pointer to `group`). The account group that the issuer and recipient accounts are part of, if they don't specify groups explicitly. This allows accounts to be specified as just *name* instead of *group:name*. Default: "yooniversal".
- `replaces` (pointer to `rawIOU`). The IOU that this IOU replaces. Default: -1, meaning it doesn't replace anything.

atomicIOU

Atomic IOUs are the raw IOUs expanded out into bilateral, non-repeating IOUs. Atomic IOUs are also created to keep track of interest payments. All atomic IOUs can be reconstructed from the raw IOUs and the interest rates.

- `amt` (real). The amount of the atomic IOU.
- `from` (pointer to `account`).[6] The account issuing the IOU.
- `to` (pointer to `account`).[7] The account receiving the IOU.
- `when` (timestamp).[8] Date/time of the atomic IOU.
- `raw` (pointer to `rawIOU`). The raw IOU which corresponds to this atomic IOU. Automatically generated interest IOUs have this field set to null.

intRate

This data structure is used to specify an interest rate for a pair of accounts at a given time. Note that interest rates are undirected, so, e.g., `alice/bob` and `bob/alice` cannot have different interest rates.

---

[2] This field is called `recips` in the database as `to` is a mySQL reserved word.

[3] This field is called `date` in the database as `when` is a mySQL reserved word.

[4] In a future version we may switch to storing timestamps as strings in YMDHMS order, with any delimiters and with at least year and month specified. Normally that should be an interface issue, but we may decide that the granularity with which the IOU time was specified is part of the information for the raw IOU that we want to record, e.g., to be able to distinguish between "June of 2008" and "2008-06-01 00:00:00".

[5] Users could in fact choose not to set currencies explicitly and just use groups to imply different currencies. E.g., all IOUs in a group "babysitting" could implicitly be denominated in hours.

[6] This field is called `issuer` in the database as `from` is a mySQL reserved word.

[7] This field is called `recip` in the database as `to` is a mySQL reserved word.

[8] This field is called `date` in the database as `when` is a mySQL reserved word.

- `acct1` (pointer to `account`). The first account.
- `acct2` (pointer to `account`). The second account.
- `rate` (real). The interest rate, stored as a fractional annual rate, not a percent, e.g., 0.05 for 5%.
- `date` (timestamp). When this interest rate goes into effect (interest rates apply until a new entry begins). Note we allow the setting of a whole historical interest timetable, not just changing the interest rate over time.

`credit`

This data structure is used to specify credit limits (or "trust") between accounts. Credit limits are directed, i.e., they need not be symmetric for a pair of accounts.

- `issuer` (pointer to `account`). The account issuing the credit.
- `recip` (pointer to `account`). The account receiving the credit.
- `amt` (real). The credit limit.
- `cur` (pointer to `currency`). The currency for this line of credit.

3.2 Data Structures Related to Users

All accounting is done in terms of the data structures in the previous section. There is a strict conceptual separation between users and accounts. Following are data structures pertaining to users.

`user`

This data structure stores usernames and passwords for users.

- `username` (string). Username with which the user accesses the Yootles system. Note that there need be no connection between this name and the names of accounts the user can access.
- `passwd` (string). The user's password, stored as an md5 hash of password plus salt.

`aliastype`

An alias type is an attribute of a user specifying another identifier (besides username) for that user. For example, "real name" or "email address" are alias types.

- `code` (string). Symbol for this alias type; one of {username, realname, email, yahoo, aol, msn, phone, 4info, icq, google, facebook}.
- `name` (string). Name for this type, e.g., "Yahoo Messenger".
- `desc` (string). Description.

`alias`

The `alias` data structure gives an attribute/value pair for a user (where the attribute is given by `aliastype`) in order to specify an alternate identifier for the user, such as Facebook ID or email address.[9]

[9] In a future version this could be generalized to arbitrary *attribute*:*value* pairs on users, such as for user settings like timezone or preferred date format.

- `user` (pointer to `user`). The user.
- `type` (pointer to `aliastype`). The alias type.
- `alias` (string). The actual alias of the specified type, e.g., "bob@bob.com", "12345", "Bob Smith", or "bobmeister12".

access

The `access` data structure encodes the ways in which a given user is allowed to access a given account.

- `user` (pointer to `user`). The user.
- `acct` (pointer to `account`). The account.
- `root` (boolean). Whether this user can change these flags for other users.
- `view` (boolean). Whether this user can view all the IOUs for this account.
- `ctrl` (boolean). Whether this user can issue IOUs from this account. This defaults to true for all users and all accounts.
- `main` (boolean). Whether this is the main account for this user (note that `main` implies `mine`).[10]
- `mine` (real). The fraction of this account's net balance that should be included in this user's net balance; typically 0 or 1.
- `ntfy` (boolean). Whether this user should get notified about IOUs for this account.

[10] In the database a user's `main` account is stored in the `user` table for performance reasons.

4 The Yootles API

Yootles API calls are made either by a specific Yootles user who specifies their Yootles username and (hashed, salted) password with each call, or on behalf of a specific Yootles user by a trusted application that provides an application-specific user alias and a (hashed, salted) application key with every call.

In both cases, API calls are REST-style over HTTP. Contrary to true REST, the HTTP status code is always set to 200 (success) with any application-level errors indicated in the API output (see Section 4.3 [Output], page 16).[1] Standard URI encoding is used in the URLs. Spaces may be mapped either to `"+"` or `"%20"`. Alphanumeric characters as well as certain characters such as `":"` and `"_"` need not be escaped. The character `"+"`, of course, must be escaped since otherwise it will be decoded as a space.

In the current beta period the API can be accessed at the following URL instead of yootles.com:

```
http://is002.yrl.re4.yahoo.com
```

4.1 API Calls by Users

The following is a template for an API command when the caller knows the Yootles username (we call this the invoking user) and password:

```
http://yootles.com/api?cmd=COMMAND
                      &invoker=USERNAME
                      &timestamp=TIMESTAMP
                      &key=KEY
```

with additional *attr=val* pairs making up the arguments for the command. For example, here is an example call to issue a simple IOU from Alice to Bob (see Section 5.8 [owe], page 30):

```
http://yootles.com/api?cmd=owe&amt=5&from=alice:ac&to=bob:b&why=lunch
                      &invoker=alice
                      &timestamp=1179548280
                      &key=289fe8ab2b3c2c19
```

In the next chapter we present API calls in an abstracted form. For example, we write the above as

```
alice> owe(amt=5, from=alice:ac, to=bob:b, why=lunch)
```

or more generally, as

```
USERNAME> COMMAND(attr=val, attr2=val2, ...)
```

This indicates that COMMAND is being called with the given named parameters and that the call is being invoked by USERNAME.

The additional fields (`key` and `timestamp`, as well as `app`, described in the next subsection) are left implicit in the abstracted form. The `timestamp` field gives the unixtime when the call was issued and the `key` field must be calculated as

[1] This choice was made to accommodate simple implementations of the API (such as perl scripts using LWP::Simple), but we may move closer to true REST in the future. Please send feedback about this to dreeves@yootles.com.

`KEY` = md5(`USERNAME + PASSWD + TIMESTAMP`)

where `PASSWD` is the user's password (and '`+`' is string concatenation). By hashing the timestamp, and not allowing old timestamps, the system limits replay attacks.

4.2 API Calls by Trusted Applications

The above suffices for applications that know usernames and passwords. Any user can, for example, use the API with calls like the above to interact with their own accounts and IOUs programmatically. We also allow access by trusted applications that do not (and cannot) know individual users' Yootles passwords but are nonetheless allowed to perform actions on behalf of users. The Yootles system maintains a mapping from identifiers on various services (including Facebook IDs, Yahoo Messenger and AIM screen names, email addresses, phone numbers for SMS, etc.) so that a trusted application need only provide the username or other identifier for its own service. For example, on Facebook if you have the numerical Facebook ID for a user, make the API call as follows:

```
http://yootles.com/api?cmd=COMMAND
                       &invoker=facebook:FACEBOOKID
                       &app=APPNAME
                       &timestamp=TIMESTAMP
                       &key=KEY
```

Anyone can apply for an application name (`APPNAME`) and key (`APPKEY`) by emailing api@yootles.com. The `key` field above is then calculated as

`KEY` = md5(`FACEBOOKID + APPKEY + TIMESTAMP`).

Since the application uses Facebook's authentication and is trusted, the Yootles password for the user on whose behalf the call is being made need not be provided. Note that the Facebook application must first create a Yootles user (see Section 5.2 [addusr], page 21) or associate the Facebook ID with an existing Yootles user (see Section 5.4 [alias], page 23). It may use any convention it chooses (e.g., FirstnameLastname); the `addusr` command indicates availability of usernames. Users can later change their usernames (see Section 5.1 [usr], page 19), but a user using the Facebook application exclusively need never even know what their Yootles username is.[2]

Other applications are analogous to the Facebook example. For example, the messenger bot application makes API calls of the form

```
http://yootles.com/api?cmd=COMMAND
                       &invoker=yahoo:YAHOOID
                       &app=yimbot
                       &timestamp=TIMESTAMP
                       &key=KEY
```

with `KEY` = md5(`YAHOOID + APPKEY + TIMESTAMP`). Besides Facebook and Yahoo Messenger, there are additional `invoker` types for other interfaces including an email bot (`"email"`) and an SMS bot (`"4info"`).

Finally, we allow the API to be called with the same consistent syntax even for calls by users instead of applications. Namely,

[2] The `request` command (see Section 5.5 [request], page 25) allows a Facebook user to learn their Yootles username and password should they want to log in to another Yootles application.

```
http://yootles.com/api?cmd=COMMAND
                      &invoker=username:USERNAME
                      &app=raw
                      &timestamp=TIMESTAMP
                      &key=KEY
```

and in fact the slightly simpler syntax given in the previous section is syntactic sugar for the above more general format.

4.3 API Output

If you specify an additional field, `output`, you can choose the output format from the set {`xml`, `json`, `perl`, `lisp`, `mma`, `php`} with `xml` the default.[3] These and other output formats are described in the next subsection.

All API calls return a set of labeled values, including a status code and a human-readable message. Formatted as JSON, API output will look like this:

```
{ "status"  : CODE,
  "message" : MESSAGE,
  ... additional "attr":"val" pairs ... }
```

`CODE` is one of the following:

- 200 – Success.
- 400 – Syntax error—malformed or incorrect arguments.
- 401 – Authorization error—invoking user not authorized to perform the operation.
- 402 – General error or exception.
- 404 – User/account/etc. not found.
- 408 – Request timed out.
- 500 – Fatal error.
- 501 – Not yet implemented.

`MESSAGE` is some human-readable output for the command (useful for command line interfaces such as the messenger bot, and also providing an explanation in case of error). After that come additional attribute/value pairs specific to the individual command, if any. Arguments for commands and their output are detailed in the next chapter.

4.3.1 Output Formats

Following is a hypothetical response that captures all the structure of the various Yootles API responses, translated into several formats. All except YAML are currently supported.[4]

JSON

```
{ "status"  : 200,
  "message" : "hello world",
  "foo"     : [{"a":1, "b":2}, {"a":10, "b":20}],
  "bar"     : {"x":"one", "y":"two", "z":"three"} }
```

[3] We will likely change the default to `json` soon.

[4] Send us feedback about your preferred output formats to dreeves@yootles.com.

Perl

The following perl expression can be directly assigned to a hashref variable. The perl module Data::Dumper generates data in the following form, given a hashref.

```
{ "status"  => 200,
  "message" => "hello world",
  "foo"     => [{"a"=>1, "b"=>2}, {"a"=>10, "b"=>20}],
  "bar"     => {"x"=>"one", "y"=>"two", "z"=>"three"} }
```

XML

Using the perl module XML::Simple with options `NoAttr` and rootname "ytl", the following XML corresponds one-to-one with the perl expression above. Note that the XML output cannot distinguish between a list containing just one element and the element itself. Also, if a list is empty then the XML output will not include the tag for that list at all.

```
<ytl>
  <status>200</status>
  <message>hello world</message>
  <foo> <a>1</a> <b>2</b> </foo>
  <foo> <a>10</a> <b>20</b> </foo>
  <bar> <x>one</x> <y>two</y> <z>three</z> </bar>
</ytl>
```

Mathematica

```
{ "status"  -> 200,
  "message" -> "hello world",
  "foo"     -> {{"a"->1, "b"->2}, {"a"->10, "b"->20}},
  "bar"     -> {"x"->"one", "y"->"two", "z"->"three"} }
```

Lisp (S-expression)

```
( ("status" . 200)
  ("message" . "hello world")
  ("foo" . ((("a" . 1) ("b" . 2)) (("a" . 10) ("b" . 20))))
  ("bar" . (("x" . "one") ("y" . "two") ("z" . "three"))) )
```

PHP

```
array('status' => 200,
      'message' => "hello world",
      'foo' => array(array('a'=>1, 'b'=>2), array('a'=>10, 'b'=>20)),
      'bar' => array('x'=>"one", 'y'=>"two", 'z'=>"three") )
```

YAML

```
---!ytl
status: 200
message: hello world
foo:
  - a: 1
```

```
    b: 2
  - a: 10
    b: 2
bar:
  x: one
  y: two
  z: three
```

4.4 Special Syntax for Specifying Accounts and Aliases

As described above, users and accounts are treated distinctly by the Yootles system. Users can access any number of accounts and an account can be accessed by any number of users. Each user, however, has at most one `main` account and in some applications it is convenient to refer to accounts by the primary user. Additionally, it is often convenient to refer to users by alias instead of username. The Yootles system supports macro expansion in API calls to facilitate this and avoid the need for multiple lookup calls. Specifically, it allows you to specify users by their aliases (such as Facebook ID or Yahoo ID) as well as refer to a user's main account indirectly by user.

4.4.1 Specifying users by alias

Similar to how accounts are specified by *group*:*name*, users may be specified by, for example, `yahoo:alice75` or `email:alice@alice.com` or `facebook:24601`. Anywhere a username can be specified, an alias can be substituted.

4.4.2 Bracket syntax for indirectly specifying accounts

The special syntax `[t:a]` or `[u]` expands to the main account of user `u` or the user referenced by alias `t:a`. This syntax can be used wherever an account is required. An error is returned if the specified user has no main account.

4.4.3 Macro for username of invoking user

`$INVOKER` is special syntax that expands to the actual username of the invoking user. This can be used anywhere. For example, set `grp=$INVOKER` to specify the invoker's username as the account group or use `[$INVOKER]` to refer to the main account of the invoking user, per the bracket syntax above. (This is equivalent to `[t:a]` where `t:a` is an alias for the invoking user.)

5 Yootles Commands

5.1 `usr` (updating and querying users)

The `usr` command allows changing usernames and passwords and looking up users by their aliases, such as email address or Facebook ID. Other commands allow the creation of new users (see Section 5.2 [addusr], page 21) and adding or changing aliases for users (see Section 5.4 [alias], page 23).

Synopsis

`usr`()
: returns the invoking user's username.

`usr`(`username`=*u*)
: changes the Yootles username of the invoking user to *u*, returning the previous username.

`usr`(`passwd`=*p*)
: changes the password of the invoking user to *p*, returning the previous password. (Note that `passwd` cannot be specified by applications, only users.)

`usr`(`username`=*u*, `passwd`=*p*)
: changes both the username and password, returning the previous values.

`usr`(`alias`=*t*:*a*)
: returns the username that has the given alias, *t*:*a*, where *t* is one of the following alias types:[1] {realname, email, yahoo, aol, msn, phone, 4info, icq, google, facebook}.

Examples

1: Changing your username

If the application issuing the command knows the user's password, the call is made, for example, as follows:

```
http://yootles.com/api?cmd=usr&username=madeleine
                      &invoker=valjean
                      &timestamp=1234567890
                      &key=3c7e8cc4b89de301
```

Otherwise, for a trusted application (e.g., the Yootles Facebook application which can make calls on behalf of users) that has an application key, a call like the following is made:

```
http://yootles.com/api?cmd=usr&username=madeleine
                      &invoker=facebook:24601
                      &app=facebook
                      &timestamp=123890
                      &key=fa3de669ef90ca6
```

Either of the above are expressed in abstracted form (see Chapter 4 [API], page 14) as:

[1] The type "username" is also considered an alias type though it wouldn't make sense to look up a user's username by their username.

```
valjean> usr(username=madeleine)
```

The following is an example response to the above:

```
{ "status"   : 200,
  "message"  : "Your username has been changed from valjean to madeleine.",
  "username" : "valjean" }
```

2: Changing your password

```
alice> usr(passwd=abc123)

{ "status"  : 200,
  "message" : "Your password has been changed.",
  "passwd"  : "changeme" }
```

If both username and password are specified, both previous values will be included in the output.

3: Looking up a user by alias

```
alice> usr(alias=email:bob77@yahoo.com)

{ "status"   : 200,
  "message"  : "Yootles user bob has email address bob77@yahoo.com.",
  "username" : "bob" }
```

5.2 `addusr` (creating new users)

Any user can call `addusr` to create a new user. An application creating a new user will also need to call the `alias` command (see Section 5.4 [alias], page 23) to create an alias for the new user and the `acct` command (see Section 5.6 [acct], page 26) to associate an account with the new user. The account may be an existing account (such as an account created by a friend) or a new account, which can be created with the `owe` command (see Section 5.8 [owe], page 30).

Synopsis

`addusr`(`username`=*u*)

> creates a new user with username *u* and system-generated password. If the user *u* already exists, the command will return with status code 402.

Example

```
alice> addusr(username=carol)

{ "status"  : 200,
  "message" : "User carol has been created.  Use the ‘request’ command
               to retrieve the password." }
```

Or, if the username "carol" is taken, the following response is generated:

```
{ "status"  : 402,
  "message" : "Error: User carol already exists." }
```

5.3 reg (macro for registering/bootstrapping a new user into the system)

The reg command is a macro implemented in terms of the other Yootles API commands. This is a placeholder for more complete documentation of this command.

```
# Create a new user, associate the alias, create the main account, and
#   link the main account.
# Eg, alias = facebook:24601, username = firstname_lastname,
#     account = firstname_lastname:firstname
# force defaults to 0; if force=1 then g:ac can't be specified
reg(username=u, alias=t:al, [force=f], [acct=g:ac])
  if f==0  (0 is the default for f)
    addusr(username=u) and abort if this fails
  else
    do addusr(username=u) with variations on u until it succeeds
  let u = the username of the added user
  let g:ac = u:u if acct not specified.
  alias(alias=t:al)  # NB: invoked with invoker=username:u
  owe(amt=0.001, from=g:ac, to=yooniversal:admin, why=new_user_fee, cur=ytl)
  acct(user=u, acct=g:ac, main=1, ntfy=1)
```

5.4 `alias` (creating, updating, and querying user aliases)

The Yootles system stores for each user additional aliases such as email addresses and instant messenger screen names. The `alias` command allows a user to query and update their aliases, and to add new ones. Other commands allow looking up a user by alias (see Section 5.1 [usr], page 19) and managing accounts for users (see Section 5.6 [acct], page 26).

The following parameters can be specified:

- `aliastype` – The alias type. E.g., `"email"`.
- `alias` – The alias, with alias type. E.g., `"email:alice@alice.com"`.
- `aname` – The descriptive name of the alias type, typically omitted.

Synopsis

`alias`()
: returns a hash table mapping alias types to aliases for the invoking user. See example 1 below.

`alias`(`aliastype`=*t*)
: returns the alias of type *t*, along with the descriptive name of the alias type. If no such alias type exists, the empty string is returned.

`alias`(`alias`=*t*:*a*, [`aname`=*n*])
: adds or updates alias type *t* to alias *a*. It returns the previous values, or the empty string if *t* did not already exist. If `aname` is also specified, this gives the descriptive name for the alias type (replacing the old one if it exists). See example 4 below.

Examples

1: Querying all aliases

```
alice> alias()

{ "status"  : 200,
  "message" : "Your aliases: email = alice@alice.com,
                             realname = Alice L. Canonikopolous,
                             facebook = 24601, yahoo = alice84",
  "aliases" : {"username":"alice", "email":"alice@alice.ccom",
               "realname":"Alice L. Canonikopolous", "facebook":24601,
               "yahoo":"alice84"} }
```

2: Querying a specific alias

```
alice> alias(aliastype=yahoo)

{ "status"  : 200,
  "message" : "Your Yahoo Messenger alias is alice84.",
  "alias"   : "alice84",
  "aname"   : "Yahoo Messenger ID" }
```

Note that this is also the way to query the descriptive name (`aname`) for the specified alias type.

3: Querying a username

This works exactly like in the previous example—"username" is treated like any other alias type. (Users, as opposed to trusted applications, will never need this since a user couldn't have issued the `alias` command without knowing their username.) Note that this usage of the `alias` command has the same functionality as calling `usr` with no arguments (see Section 5.1 [usr], page 19).

```
alice/24601> alias(aliastype=username)

{ "status"  : 200,
  "message" : "Your Yootles username is alice.",
  "alias"   : "alice",
  "aname"   : "Yootles username" }
```

4: Adding and modifying an alias

```
alice> alias(alias=openid:oid.org/alice, aname="Open ID")

{ "status"    : 200,
  "message"   : "New alias added: Open ID (openid) = oid.org/alice",
  "aliastype" : "",
  "alias"     : "",
  "aname"     : "" }
```

The output fields `aliastype`, `aname`, and `alias` give the previous values, or the empty string if this is a new alias.

```
alice> alias(alias=facebook:24602)

{ "status"    : 200,
  "message"   : "Facebook ID changed from 24601 to 24602.",
  "aliastype" : "facebook",
  "alias"     : 24601,
  "aname"     : "Facebook ID" }
```

5.5 `request` (email a user their username and password)

When an application creates a new user, the Yootles system generates a random password for that user, which the application never sees (since applications have application keys and can issue commands on behalf of users without knowing user passwords). A user using a particular application need never know their Yootles username or password, but if the user would like to use some other application that uses the Yootles API, the user will need to provide their username and password to that other application. The `request` command causes an email to be sent to the user with their username and password.

Synopsis

`request`()

> returns success and, as a side effect, generates email. If email could not be sent—for example, because no email alias is set (see Section 5.4 [alias], page 23)—it returns status code 402.

Example

```
alice> request()
{ "status"  : 200,
  "message" : "Your Yootles username and password has been
               sent to alice@alice.com" }
```

Known Bugs

There is only one "bug": this command has not been implemented yet. For the current alpha version, the password is just always set to `changeme` and if you actually change it and then forget it, you'll have to email help@yootles.com. Note that trusted applications, like the Facebook application, do not need to deal with user passwords.

5.6 `acct` (querying and setting access control)

The `acct` command allows management of the `access` data structure (see Section 3.2 [User Data Structures], page 12). That is, it allows the setting and querying of which users have access to which accounts. Parameters for the `acct` command correspond directly to the fields in the `access` data structure, namely: `user`, `acct`, `main`, `mine`, `ntfy`, `root`.

Synopsis

`acct`([`user`=*u*])

returns, for user *u* (default: the invoking user), the `main` account, a list of `mine` accounts (accounts for which *u* has `mine` > 0), a list of `ntfy` accounts (accounts for which *u* gets notifications), and a list of `root` accounts (accounts for which *u* has root access).

`acct`(`user`=*u*, `acct`=*a*)

returns, for each access field, user *u*'s access to account *a*.

`acct`(`acct`=*a*)

returns, for each access field, the list of users (possibly empty) who have the corresponding access to account *a*. Having `mine`>0 suffices to be included in the list of users having `mine` access.

`acct`([`user`=*u*], `acct`=*a*, <*access-settings*>)

sets user *u*'s access to account *a* according to <*access-settings*> and returns the previous access settings. The placeholder <*access-settings*> denotes any nonempty subset of the access fields (see example 5). User *u* may be omitted and will default to the invoking user.

Examples

1: Checking your own accounts

```
alice> acct()

{ "status"   : 200,
  "message"  : "You (alice) have main account alice:alc (main)",
  "main"     : "alice:alc",
  "mine"     : ["alice:alc"],
  "ntfy"     : ["aliceposse:jointfund", "alice:mom", "alice:alc"],
  "root"     : ["alice:alc", "alice:mom"] }
```

2: Checking someone else's accounts

```
alice> acct(user=bob)

{ "status"   : 200,
  "message"  : "User bob has main account bob:b.",
  "main"     : "bob:b",
  "mine"     : ["bob:b"],
  "ntfy"     : ["bob:b"],
  "root"     : ["bob:b"] }
```

3: Checking the access for a particular user/account pair

```
alice> acct(user=bob, acct=carol:c)

{ "status"  : 200,
  "message" : "User bob has the following access to the carol:c account:
               root=0, view=1, ctrl=1, main=0, mine=0, ntfy=0",
  "main"    : 0,
  "mine"    : 0,
  "ntfy"    : 0,
  "root"    : 0 }
```

4: Checking who has access to a particular account

```
alice> acct(acct=carol:c)

{ "status"  : 200,
  "message" : "User carol has carol:c as their main account.",
  "main"    : ["carol"],
  "mine"    : ["carol"],
  "ntfy"    : ["carol", "alice"],
  "root"    : ["carol", "alice"] }
```

5: Setting access for a user/account pair

This example will only succeed if Alice has root access to the carol:c account.

```
alice> acct(user=bob, acct=carol:c, ntfy=1, mine=0.25)

{ "status"  : 200,
  "message" : "User bob's access to account carol:c has been updated.",
  "main"    : 0,
  "mine"    : 0,
  "ntfy"    : 0,
  "root"    : 0 }
```

5.7 grp (querying and adding account groups)

The `grp` command is used to query the list of groups relevant to the invoking user, to query the accounts that belong to an account group, or to add new groups. The input parameters are as follows:

- `grp` – The group name. May be omitted to get a list of all relevant groups.
- `desc` – A short description of the group. This field can be specified to update the group description, or omitted if querying an existing group.

Synopsis

`grp`()

returns a list of all groups in which the invoking user owns an account (`mine`>0) or has root access on some account.

`grp`(`grp`=*g*)

returns the description of the group and a list of the member accounts. This command only succeeds if the invoking user owns an account in the group.

`grp`(`grp`=*g*, `desc`=*d*)

modifies the description of a group to *d* or creates group *g* if it does not exist. It returns the previous description and the group name *g* or the empty string if it did not exist.

Note that the description is required for adding a new group and that the command returns the previous description when modifying it. It returns the empty string for both `grp` and `desc` to indicate that a new group was created. An error is returned if a nonexistent group is queried, and the group is not created in this case (see last example).

Examples

1: Listing groups

```
alice> grp()

{ "status"  : 200,
  "message" : "Groups: jets, sharks, alice, smithfamily, aliceposse.",
  "groups"  : ["jets", "sharks", "alice", "smithfamily", "aliceposse"] }
```

2: Querying a particular group

```
alice> grp(grp=jets)

{ "status"  : 200,
  "message" : "jets (a west-side gang) has members: alice, bob, carol.",
  "grp"     : "jets",
  "desc"    : "a west-side gang",
  "members" : ["alice", "bob", "carol"] }
```

3: Creating a new group

```
alice> grp(grp=yahoos, desc="Yahoo employees")
```

```
{ "status"  : 200,
  "message" : "New group ‘yahoos’ created: Yahoo employees.",
  "grp"     : "",
  "desc"    : "" }
```

4: Modifying a group

```
alice> grp(grp=yahoos, desc="Yahoo employees and friends")

{ "status"  : 200,
  "message" : "Description updated to ‘Yahoo employees and friends’.",
  "grp"     : "yahoos",
  "desc"    : "Yahoo employees" }
```

5: Nonexistent group

```
alice> grp(grp=losers)

{ "status"  : 402,
  "message" : "Account group ‘losers’ not found. Provide name and
               description to create group.",
  "grp"     : "" }
```

5.8 `owe` (adding or modifying an IOU)

The `owe` command issues an IOU, possibly a repeating and/or multilateral IOU (see Section 1.2 [IOUs], page 1). The input fields correspond to the `rawIOU` data structure and are described in greater detail there (see Chapter 3 [Data Structures], page 10). Note that fields of type `"timestamp"` are specified in unixtime.

Synopsis

`owe`(`amt`=*a*, [`from`=*u*], `to`=*t*, [`when`=*w*], `why`=*y*, [`cur`=*c*], [`grp`=*g*], [`replaces`=*r*])

> issues a non-repeating IOU of amount *a*, denominated in currency *c* (default: yootles), from accounts *u* (default: the main account of the invoking user) to accounts *t* at time *w* (default: now), logging *y* as the reason and with *r* giving the ID of a previous IOU to replace, if any. If either account does not have a group prefix, *g* is used. The main accounts of the specified users will be used.) The output fields are given below.

`owe`(..., `rpt`=*r*, `rptunit`=*u*, [`til`=*t*])

> issues a repeating IOU, with all parameters the same as for a non-repeating IOU plus the addition of the above, specifying a payment period of *r*, measured in units *u*, and a time *t* after which no more repeating IOUs will be generated (default: *t*=-1, meaning IOUs repeat indefinitely).

The output fields for the `owe` command are as follows:

- `iou` – The ID for the new (raw) IOU.
- `num` – Number of IOUs (1 if not repeating, -1 if repeating indefinitely, never 0 or less than -1, and includes the final IOU if repeating, even if it's prorated to 0).
- `last` – Fraction at which the last IOU is pro-rated. This is always 1 for non-repeating IOUs.
- `accounts` – List of accounts involved in this IOU.
- `deltas` – A list parallel to `accounts` with the deltas on those accounts' net balances due to this IOU (counting only the first IOU if repeating).
- `atomized` – List of the atomic IOUs (but not including repeating IOUs past the initial IOU), i.e., a list of amt/from/to triples (represented as hash tables). For a normal bilateral IOU this will simply echo the IOU as it was input, except that it will evaluate the amount which may have been sent as (and is stored as) a mathematical expression. Note that some multilateral IOUs will generate atomic IOUs from an account to itself—it is up to the interface to filter these out or not (sometimes it can be helpful to see them).
- `spawn` – List of accounts newly created due to this IOU. (Both groups and accounts will be created as needed.) This list is a subset of `accounts`.

Note that the `owe` command is the only way to create new accounts (the amount may be set to zero if no actual IOU is wanted). The notification for the IOU should make clear that a new account was created so that in case of a typo the IOU can be reissued. Using an IOU to create an account has the advantage that it gets logged in the transaction history.

Example

Alice issues an IOU to a new user, Bob

The following call issues an IOU from `user:alice` (which maps to Alice's main account, e.g., `alice:alc`) to the account `alice:bob` (an account Alice is creating for her friend Bob) for 12 yootles "for lunch":

```
alice> owe(amt=12, to=bob, why="for lunch", grp=alice)

{ "status"   : 200,
  "message"  : "Your (alice's) balance with bob has decreased 12 yootles.
                You now owe bob 57 yootles.",
  "iou"      : 24601,
  "num"      : 1,
  "last"     : 1,
  "accounts" : ["alice:alc", "alice:bob"],
  "deltas"   : [-12, 12],
  "atomized" : [{"amt" : 12, "from" : "alice:alc", "to" : "alice:bob"}],
  "spawn"    : ["alice:bob"] }
```

Note that the only way to effectively delete an IOU is to issue a new `owe` command with the `replaces` field set to the ID of the old IOU and the amount set to 0 (or perhaps with the amount multiplied by zero so it's easy to see what the original amount was—this makes for a quick-and-dirty audit trail). The advantage of this approach to voiding old IOUs is that the audit trail for every "deleted" IOU ends with an active IOU. That means no IOU can ever go fully out-of-sight-out-of-mind. This is important given the liberal access policy; remember that anyone who has permission to control a pair of accounts (by default, everyone) can replace IOUs between those accounts.

5.9 `tran` (querying existing IOUs)

The `tran` call allows retrieval of transaction histories, i.e., a list of past IOUs. Each parameter (except `atomize`) specifies a filter to apply to the IOUs, restricting the set of returned results. If a parameter has no default value then leaving that parameter unspecified omits the corresponding filter. With no parameters specified, `tran` returns every IOU in all of the invoking user's groups (see Section 5.7 [grp], page 28).

The parameters for `tran` are as follows:

- `acct1` – Only return IOUs involving a particular account, specified as *group:name*.
- `acct2` – This works identically to `acct1`. Specify both `acct1` and `acct2` to get transactions involving a pair of accounts.
- `grp` – Only return IOUs involving a particular group.
- `mine` – Only return IOUs involving accounts belonging to the invoking user (i.e., `mine` > 0).
- `start` – The time of the earliest IOU to return (unixtime).
- `end` – The time of the latest IOU to return (unixtime). Default: -1, meaning the time of the last *explicit* IOU, i.e., the maximum of all the IOU times and end times for repeating IOUs, if specified. For raw IOUs this is equivalent to omitting the `end` filter altogether. But because of indefinitely repeating IOUs, it is not well-defined to ask for all atomic IOUs after a given time.
- `all` – Only return IOUs that have been replaced (1 means yes). Default: 0, meaning IOUs will be filtered to exclude replaced IOUs.
- `iou` – Only return the IOU with the specified ID, plus the IOUs that the specified one replaced. Note, however, that any replaced IOUs will be filtered out unless `all`=1. This also means there is no way to query a single IOU that has been replaced—you can only query it along with the chain of subsequent IOUs that replaced it.
- `atomize` – Show atomized IOUs (1 means yes). Default: 0, meaning no, show raw IOUs which may be repeating and/or multilateral IOUs.
- `limit` – Limit the IOUs returned to at most the specified number, starting with the most recent. Default: infinity.
- `offset` – Skip the specified number of most recent transactions. Default: 0. The parameters `limit` and `offset` are used to retrieve transactions in batches. Note that `offset` is zero-based so that, for example, calling `tran` with `limit`=10 and `offset`=100 returns the 101st through the 110th IOU (in reverse chronological order).

Synopsis

`tran`([`acct1`=*a*], [`acct2`=*b*], [`grp`=*g*], [`mine`=*m*], [`start`=*s*], [`end`=*e*], [`all`=*d*], [`iou`=*i*])

returns a list of raw IOUs which involve an account in one of the invoking user's groups subject to the following constraints (assuming the corresponding parameter was specified):

- the IOU involves account *a*
- the IOU involves account *b*
- the IOU involves an account in group *g*

- at least one of the accounts has `mine` greater than or equal to *m* for the invoking user
- the date of the IOU is on or after *s*
- the date of the IOU is on or before *e* (with the default of *e*=-1 interpreted as explained above)
- the IOU has not been replaced (if *d*=0, which is the default, otherwise no constraint—both replaced and nonreplaced IOUs are returned)
- the IOU has ID *i* or was eventually replaced by *i*

For repeating IOUs only the nominal (i.e., initial) IOU date is considered when comparing to *s* and *e*. This means that, for example, a repeating IOU that starts in 2007 and continues in 2008 will not be included in a query for raw IOUs in 2008.

`tran`(..., `atomize`=1)

is the same as above, in fact returning the exact same IOUs, but in atomized form, showing only pairwise, nonrepeating IOUs (see Chapter 3 [Data Structures], page 10). Note that by atomizing post hoc, some atomic IOUs can be returned that would otherwise be precluded by the filters. For example, a multilateral IOU from Alice and Bob to Carol is atomized into separate IOUs from Alice to Carol and Bob to Carol. When IOUs involving Alice are requested, the atomic IOU from Bob to Carol is returned because it is part of a raw IOU involving Alice. Only the `end` field applies directly to the atomized IOUs.

`tran`(..., [`limit`=*n*], [`offset`=*s*])

is the same as above but only returns, of the IOUs that would have been returned otherwise, the *s*+1st through *s*+*n*th results.

When `atomize`=0 (the default), the output field `rtran` (for raw transactions) is returned, which is a list of hash tables, each with the following fields, corresponding roughly to the fields of the `rawIOU` data structure (see Chapter 3 [Data Structures], page 10):

- `iou` – The ID of the IOU.
- `amt` – The amount, a mathematical expression, the same way it was specified in the corresponding `owe` command.
- `from` – The account(s) issuing the IOU.
- `to` – The account(s) receiving the IOU.
- `fromref` – Mirrors `from` but with each account replaced by, in square brackets, the alias of the user who has that account as their main account, if such a user exists. The alias to use is determined by the one the invoker was identified by.
- `toref` – Analogous to `fromref`.
- `when` – The date/time of the IOU (specified in unixtime).
- `why` – The reason for the IOU.
- `rpt` – The repeat interval.
- `rptunit` – Repeat unit.
- `til` – End time for repeating IOUs.

- `cur` – Currency code.
- `grp` – Account group for any accounts in `from` and `to` with no group specified.
- `replaces` – The ID of the IOU that this one replaces; -1 means none.

Similarly, when `atomize`=1, the output field `atran` (for atomic transactions) is returned, which is a list of hash tables each with the following fields:

- `iou` – The ID of the (raw) IOU this atomic IOU is part of.
- `amt` – The amount, a real number.
- `from` – The account issuing the IOU.
- `to` – The account receiving the IOU.
- `fromref` – The alias of the user whose main account is the account in `from`.
- `toref` – Analogous to `fromref`.
- `when` – The date/time (specified in unixtime).
- `why` – The reason for the IOU, taken from the corresponding raw IOU and annotated to indicate repeating payments and whether it was prorated.
- `cur` – Currency code.

Finally, an additional output field `count` gives the total number of entries (raw or atomic IOUs) that would be returned with `offset`=0 and `limit`=*infinity* (the defaults).

Examples

1: Viewing all (raw) IOUs in any of your groups

```
alice> tran()

{ "status"  : 200,
  "message" : "There are 100 IOUs in your (alice's) groups.",
  "count"   : 100,
  "rtran"   : [{"iou":234, "amt":10, "from":"alice:alc+bob", "to":"bob",
                "fromref":"[alice]+[bob]", "toref":"[bob]",
                "when":1234567890, "why":"for lunch", "rpt":-1,
                "rptunit":"year", "til":-1, "cur":"usd",
                "grp":"yooniversal", "replaces":-1}, ... ] }
```

2: Viewing all your (atomized) IOUs with a specific account

```
alice> tran(acct1=yooniversal:bob, atomize=1)

{ "status"  : 200,
  "message" : "Account yooniversal:bob has 150 IOUs.",
  "count"   : 150,
  "atran"   : [{"iou":234, "amt":5, "from":"alice:alc",
                "to":"yooniversal:bob", "fromref":"[alice]", "toref":"[bob]",
                "when":1234567890, "why":"for lunch", "cur":"usd"}, ... ] }
```

Known Bugs

Currently the base set of IOUs is any IOU involving any of the invoking user's accounts (accounts for which the invoking user has `mine` access) rather than all IOUs in all of the invoker's groups (see Section 5.7 [grp], page 28). Thus the `grp` field is currently treated specially. It both restricts the results to include only IOUs involving accounts in the specified group, and it also *extends* the returned results to include IOUs that do not include accounts belonging to the invoking user.

Additionally, the *group* prefix can be omitted currently if a group is specified in the `grp` field. This overloaded use of the `grp` field is deprecated and will soon go away. You should always specify *group:account* explicitly in the `acct1` and `acct2` fields.

Finally, `fromref` and `toref` do not behave as advertized. In particular, they only work when there is a single account in the corresponding `from` or `to`.

5.10 `bal_old` (querying balances)

The following has been deprecated. See the new `bal` *command documentation in the next section.*

The `bal` command queries an account's balance with a specified other account in a specified currency, or, if another account is not specified, `bal` returns the primary account's balances with every other account for which the balance is nonzero. The user calling `bal` must have `view` access on at least one of the two accounts. By specifying a group but not an account, you can see the balances with all accounts in a group.

The input parameters and their defaults are as follows:

- `acct1` – The account whose balance is to be queried, specified as *group:name*, or, if `grp` is specified, just *name*. Default: the main account of the invoking user.
- `acct2` – A specific account to check balances with.
- `asof` – Return the balance(s) as of this time, specified in unixtime. Default: now, i.e., when the command is issued.
- `grp` – Group name to prepend to `acct1` and `acct2`. If `acct2` is not specified, `grp` specifies the group within which to return balances.
- `cur` – Currency code.

There are two primary ways to use the `bal` command: specifying an `acct2` field or not. If `acct2` is not specified then `bal` returns `acct1`'s balance with each of a set of accounts in the specified currency. A positive balance means `acct1` is owed, a negative balance means `acct1` owes, and any account for which a balance is not included has balance zero. If `acct2` is specified then `bal` returns `acct1`'s balance with `acct2` for each *currency* (excluding currencies in which that balance is zero).

Synopsis

`bal`([`acct1`=*a*], [`asof`=*t*])

returns account *a*'s net balance for each currency as of time *t*.

`bal`([`acct1`=*a*], `cur`=*c*, [`asof`=*t*])

returns all nonzero balances in currency *c* for account *a* as of time *t*.

`bal`([`acct1`=*a*], `cur`=*c*, `grp`=*g*, [`asof`=*t*])

returns, for each account *i* in group *g*, the balance that account *a* has (as of time *t*) with *i* in currency *c*. Note the dual use of `grp`—it specifies the group to prepend to *a* (if necessary) and also indicates that the returned balances should be limited to transactions with accounts in group *g*.

`bal`([`acct1`=*a*], `acct2`=*b*, `cur`=*c*, [`grp`=*g*], [`asof`=*t*])

returns the balance in currency *c* between accounts *a* and *b* as of time *t*. The group *g*, if specified, will be prefixed to *a* and *b* if they do not specify a group.

`bal`([`acct1`=*a*], `acct2`=*b*, [`grp`=*g*], [`asof`=*t*])

returns, for each currency *i*, the balance that account *a* has (or had at time *t*) with account *b* in currency *i*. The group *g*, if specified, will be prefixed to *a* and *b* if they do not specify a group.

Examples

1: Querying net balance

```
alice> bal()

{ "status"     : 200,
  "message"    : "Your (alice:alc's) net balance is 10 usd and -20 ytl.",
  "currencies" : ["usd", "ytl"],
  "balances"   : [10, -20] }
```

2: Querying your balance in a group

The following call returns the balances between Alice's main account and each of the accounts in the `jets` group:

```
alice> bal(grp=jets, cur=ytl)

{ "status"   : 200,
  "message"  : "Your (alice:alc's) net yootles balance in group jets is -3.",
  "accounts" : ["jets:bob", "jets:carol"],
  "balances" : [-23, 20] }
```

3: Querying your balance with a specific account, in all currencies

If in the following example `acct2` was specified as `jets:bob` then the `grp` field would be superfluous. In fact, with `acct2=jets:bob` and `grp=sharks`, the `grp` field would be entirely ignored.

```
alice> bal(acct2=bob, grp=jets)

{ "status"     : 200,
  "message"    : "Your (alice:alc's) balances with jets:bob are: 12 usd, 3 ytl.",
  "currencies" : ["usd", "ytl"],
  "balances"   : [12, 3] }
```

4: Querying the balance of another account

Note that in this example `currencies` and `balances` are lists (as always when `acct2` is specified) even though there is only one currency requested.

```
alice> bal(acct1=bob, acct2=carol, cur=ytl, grp=jets, asof=1201150800)

{ "status"     : 200,
  "message"    : "bob's balance with carol (in group jets) as
                  of 2008.01.24 is +7 ytl.",
  "currencies" : ["ytl"],
  "balances"   : [7] }
```

Known Bugs

Specifying `grp` with no `acct2` specified (as in Example 2 and the third synopsis case) should limit balances to IOUs with accounts in the specified group. Currently this restriction is not respected and the overall balance is returned.

5.11 `bal` (querying balances)

The `bal` command works by selecting a set of *atomic* IOUs and then, for each account involved in any of those IOUs, computes the balance for that account within that set of IOUs. In addition, the invoker's net balance (computed using the invoker's `mine` fraction for each account) within the set of IOUs is returned. The initial set of IOUs is, just as in the `tran` command (see Section 5.9 [tran], page 32), all those which include one of the invoker's groups (see Section 5.7 [grp], page 28). As with `tran`, each parameter for `bal` filters this set of IOUs. The `bal` command returns a hash mapping each account to its balance within the filtered set of IOUs. For example, specifying the `grp` field gives everyone's net balance within a group. And specifying an account gives every other account's net balance with that account.

The `bal` command takes the following parameters:

- `acct1` – Show this account's net balance and everyone's balance with it.
- `acct2` – Assuming `acct1` is also specified, return the balance between the two accounts.
- `grp` – Only show balances within the given group.
- `asof` – Show balances as of a given time in the past or future. This is analogous to the `end` field in the `tran` command.
- `cur` – Show balances in the given currency. This field must be specified.

Synopsis

`bal`([`acct1`=*a*], [`acct2`=*b*], [`grp`=*g*], [`asof`=*t*], `cur`=*c*)

filters the atomic IOUs in the invoker's groups according to the following constraints (if specified):

- the IOU involves account *a*
- the IOU involves account *b*
- the IOU involves an account in group *g*
- the date of the IOU is no later than *t*
- the currency of the IOU is *c*

and then returns for each account included in any of the IOUs, the account's net balance within that set of IOUs.

In addition to returning a hash mapping accounts to balances, `bal` returns a `netbal` field with the sum of all the balances in the hash weighted by the invoker's `mine` fraction for the corresponding account.

Examples

1: Querying all account balances and invoker's net balance

```
alice> bal(cur=ytl)

{ "status"  : 200,
  "message" : "Your (alice's) net balance across all your groups is 97 ytl.",
  "bal"     : {"alice:alc" : 100, "bob:bob": -100, "carol:crl" : 0,
               "jets:alc" : -3, "jets:bob" : -20, "jets:carol" : 23},
  "netbal"  : 97 }
```

2: Querying everyone's balances within a given group

```
alice> bal(grp=jets, cur=ytl)

{ "status"  : 200,
  "message" : "Your (alice's) net balance in group jets is -3 ytl.",
  "bal"     : {"jets:alc" : -3, "jets:bob" : -20, "jets:carol" : 23},
  "netbal"  : -3 }
```

3: Querying everyone's balances with a given account

```
alice> bal(acct1=alice:alc, cur=ytl)

{ "status"  : 200,
  "message" : "Account alice:alc has net balance 100 ytl.",
  "bal"     : {"alice:alc" : 100, "bob:bob" : -100, "carol:crl" : 0},
  "netbal"  : 100 }
```

4: Querying a pairwise balance between two accounts

```
alice> bal(acct1=alice:alc, acct2=bob:bob, cur=ytl)

{ "status"  : 200,
  "message" : "Account bob:bob owes alice:alc 100 ytl.",
  "bal"     : {"alice:alc" : 100, "bob:bob" : -100},
  "netbal"  : 100 }
```

5: Querying balances as of a given time

```
alice> bal(acct1=alice:alc, acct2=bob:bob, cur=ytl, asof=1201150800)

{ "status"  : 200,
  "message" : "As of 2008.01.24, bob:bob owes alice:alc 100 ytl.",
  "bal"     : {"alice:alc" : 100, "bob:bob" : -100},
  "netbal"  : 100 }
```

Known Bugs

This command is currently implemented as a macro on top of `tran` which makes it slow. Also, `asof` must currently be specified as `end`. Finally, `cur` can currently be omitted and the balances will sum across currencies as if they all had 1:1 exchange rates.

5.12 `intr` (querying and setting interest rates)

The `intr` command allows a user who has `view` access for either of two accounts to view the interest rate(s) between those accounts and for a user with `ctrl` access for either of two accounts to add or change interest rates between them. In addition to the standard parameters identifying the user making the call, `intr` is called with the following parameters:

- `acct1` – The first account. Default: the main account of the invoking user.
- `acct2` – The second account. (Note that the order of `acct1` and `acct2` is irrelevant as interest rates are symmetric.)
- `rate` – The annual interest rate, expressed as a real number fractional rate, not a percentage (e.g., 0.05 for 5%).
- `date` – The date/time, specified in unixtime.

Interest rates between accounts always apply to all currencies. Different currencies cannot have different interest rates—instead, users can create new accounts to get different interest rates.

Synopsis

`intr`([`acct1`=*a*], [`date`=*d*])

> returns a hash table mapping accounts to the interest rates that account *a* had set for those accounts on date *d* (i.e., for the most recent stored date less than or equal to *d*). Account *a* defaults to the main account of the invoking user and date *d* defaults to *now*, the time the call was issued.

`intr`([`acct1`=*a*], `acct2`=*b*)

> returns the list of date/rate pairs between the accounts *a* and *b*.

`intr`([`acct1`=*a*], `acct2`=*b*, `date`=*d*)

> returns the interest rate on date *d* and the date when that rate went into effect.

`intr`([`acct1`=*a*], `acct2`=*b*, [`date`=*d*], `rate`=*r*)

> sets the interest rate between accounts *a* and *b* to *r*, effective on date *d* (or *now* if `date` is not specified). If *d* exactly matches an existing entry (and rate *r* is different from that entry's rate), the existing entry will be replaced. If *r* is the same as the rate in the preceding entry (preceding by date), the command is a no-op—the new interest rate is not inserted.[1] The date and rate of the preceding entry (or the replaced entry, if the dates matched) are returned.

Examples

1: Checking your current interest rates with everyone

```
alice> intr()

{ "status"  : 200,
  "message" : "Interest rates: bob:b = 5%, jets:carol = 12%.",
  "intr"    : { "bob:b":0.05, "jets:carol":0.12 } }
```

[1] If you want an identical interest rate inserted, say, 5% on Feb 1 to follow an existing rate of 5% on Jan 1, call the `intr` command with a date of Feb 1 and a rate of 0% (or anything not equal to 5%) and then call it again with a date of Feb 1 and a rate of 5%.

2: Setting a new interest rate with someone

```
alice> intr(acct1=alice:alc, acct2=bob:b, rate=0.06, date=1192809600)

{ "status"  : 200,
  "message" : "Interest rate with bob:b changed to 6%,
               effective 2007-10-19 through 2007-10-20.",
  "date"    : 1192723200,
  "rate"    : 0.05 }
```

3: Checking your historical interest rates with someone

```
alice> intr(acct2=bob:b)

{ "status"  : 200,
  "message" : "Your interest rate with bob:b has varied
               between 5% and 6% and is now 6%.",
  "intr"    : [{"date":123456789, "rate":0.05},
               {"date":123500000, "rate":0.06}] }
```

4: Checking your interest rate with someone on a particular date

```
alice> intr(acct2=carol:c, date=1201323600)

{ "status"  : 200,
  "message" : "Your interest rate with carol:c on 2008-01-26 was 6%
               which went into effect on 2007-11-01.",
  "date"    : "1193889600",
  "rate"    : 0.06 }
```

Known Bugs

There is only one "bug": this command has not been implemented yet.

5.13 `cred` (querying and setting credit limits)

The `cred` command sets and queries credit limits between a (directed) pair of accounts. The invoking user must have `view` access (see Section 1.8 [Access Control], page 4) on either of the two accounts to see the credit limit between them. When extending credit between accounts the invoking user must have `ctrl` access on the account extending credit, and accounts that don't already exist are created. Any currency or account not listed in the results has an implied credit limit of zero. Input parameters are as follows:

- `from` – The account issuing the credit.
- `to` – The account receiving the credit; if this account doesn't exist it is automatically created.
- `amt` – The amount of credit; must be non-negative.
- `cur` – Currency code.

Synopsis

`cred`(`to`=*a*, `cur`=*c*)

returns a hash table mapping accounts to the credit limits that those accounts have extended to account *a* in currency *c*.

`cred`(`from`=*a*, `cur`=*c*)

returns a hash table mapping accounts to the credit limits that account *a* has extended credit to those accounts in currency *c*.

`cred`(`from`=*a*, `to`=*b*, `cur`=*c*)

returns the credit limit from *a* to *b* in currency *c*.

`cred`(`from`=*a*, `to`=*b*)

returns a hash table mapping currencies to credit limits from *a* to *b* in each currency.

`cred`([`from`=*a*], `to`=*b*, `amt`=*x*, `cur`=*c*)

changes the credit limit (from *a* to *b*) to *x* in currency *c* and returns the previous amount of credit. Account *a* defaults to the main account of the invoking user. (The invoking user must have `ctrl` access on account *a*.)

Examples

1: Offering someone credit

```
alice> cred(from=alice:alc, to=bob:b, amt=7, cur=ytl)

{ "status"  : 200,
  "message" : "The account bob:b now has a credit limit of
               7 yootles (was 0) with alice:alc.",
  "cred"    : 0 }
```

2: Checking your credit limit with another account

```
alice> cred(from=alice:alc, to=bob:b)
```

```
{ "status"  : 200,
  "message" : "The account bob:b has a credit limit of
               7 yootles and 100 dollars with alice:alc.",
  "cred"    : {"ytl":7, "usd":100} }
```

3: Checking your total available credit for a particular currency

```
alice> cred(to=alice:alc, cur=ytl)

{ "status"  : 200,
  "message" : "The account alice:alc has a total of 99 yootles of credit.",
  "cred"    : {"bob:b":11, "jets:carol":88} }
```

4: Checking the total credit you have extended in a particular currency

```
alice> cred(to=alice:alc, cur=ytl)

{ "status"  : 200,
  "message" : "The account alice:alc has extended 33 yootles of credit.",
  "cred"    : {"bob:b":11, "jets:carol":22} }
```

Known Bugs

There is only one "bug": this command has not been implemented yet. All credit limits are zero. But note that this does not restrict the IOUs that can be issued, it just means that any balances exceed the credit limits.

5.14 `cur` (managing currencies)

The `cur` command is used to query or change information about a currency, or to add a new currency. The parameters for the `cur` command are as follows:

- `code` – The short symbolic name for a currency, such as usd for US dollars or ytl for yootles.
- `name` – The full name of the currency.
- `desc` – A short description of the currency.

The output fields are the same as the input fields except in the case that `cur` is called with no parameters, in which case the only output field is `cur`.

Synopsis

`cur`()

returns a simple list of available currencies (see example 1).

`cur`(`code`=*c*)

returns the code, name, and description if the currency exists. Otherwise, it returns an error (`status` = 404) and the empty string in the `code` field.

`cur`(`code`=*c*, `desc`=*d*)

changes the description for the currency, returning the original code, name, and description. An error like above is returned if the currency does not exist.

`cur`(`code`=*c*, `name`=*n*)

changes the name for the currency, returning the original code, name, and description. An error like above is returned if the currency does not exist.

`cur`(`code`=*c*, `name`=*n*, `desc`=*d*)

changes the name and description for the currency, returning the original code, name, and description. If the currency does not exist it is created and empty strings are returned for all three fields.

Examples

1: Listing all available currencies

```
alice> cur()

{ "status"  : 200,
  "message" : "Currencies: ytl, usd, inr, can, beer.",
  "cur"     : ["ytl", "usd", "inr", "can", "beer"] }
```

2: Querying the name and description for "ytl"

```
alice> cur(code=ytl)

{ "status"  : 200,
  "message" : "Yootles (ytl): A unit of utility or happiness.
               Also spelled utils or utiles.",
  "code"    : "ytl",
```

```
  "name"    : "Yootles",
  "desc"    : "A unit of utility or happiness.
               Also spelled utils or utiles." }
```

3: Creating a new currency, "goats"

```
alice> cur(code=goat, name="Goats", desc="Actual live goats")

{ "status"  : 200,
  "message" : "New currency ‘goat’ (Goats) created.",
  "code"    : "",
  "name"    : "",
  "desc"    : "" }
```

4: Modifying a currency

```
alice> cur(code=goat, desc="Actual number of live goats")

{ "status"  : 200,
  "message" : "Description updated to ‘Actual number of live goats’.",
  "code"    : "goat",
  "name"    : "Goats",
  "desc"    : "Actual live goats" }
```

5: Nonexistent currency

```
alice> cur(code=nuggets)

{ "status"  : 402,
  "message" : "Currency code ‘nuggets’ not found.
               Provide name and description to create currency.",
  "code"    : "" }
```

5.15 `merge` (merging/renaming accounts)

The `merge` command takes parameters `old` and `new` specifying two account names, with `old` defaulting to the main account of the user issuing the command. An optional parameter `grp` gives the group for the accounts if they are not specified in `old` and `new`. The following steps are performed (where we assume that `old` is "old:jekyll" and `new` is "new:hyde"):

- Check that the user has `root` access on "old:jekyll" and `ctrl` access on every account that has issued "old:jekyll" credit or that "old:jekyll" owes (has a negative balance with). If not, abort with status code 402.[1]
- Create the account "new:hyde" if it doesn't exist (including creation of the group "new" if needed).
- For each nonzero incoming and outgoing credit limit with "old:jekyll", create the same credit with "new:hyde" and set the credit with "old:jekyll" to zero.
- For each interest rate returned by the `intr` command (needs to be called once to get the users, then for each user to get the list of historical interest rates), create the same interest rate with "new:hyde".
- For each account *a* with whom "old:hyde" has nonzero balance *b*:
  - Zero the balance by issuing an IOU of *b* from "old:jekyll" to *a* with comment "[renaming old:jekyll -> new:hyde]".
  - Issue an IOU of *b* from *a* to "new:hyde" with the same comment.

Synopsis

`merge`(`old`=*x*, `new`=*y*, [`grp`=*g*])

follows the above procedure to merge account *x* into account *y*. (If *x* or *y* do not specify groups, use group *g*.)

Example

```
alice> merge(old=old:jekyll, new=new:hyde)

{ "status"  : 200,
  "message" : "Account old:jekyll merged into new:hyde." }
```

Known Bugs

There is only one "bug": this command has not been implemented yet.

5.16 Acknowledgments

Thanks to Sharad Goel, Dave Morris, Meg West, and Laurie Reeves for providing valuable comments and fixing errors in early drafts of this documentation.

[1] In the future we may want to be more forgiving. This can be done if the user has the same access rights for "new:hyde" as for "old:jekyll" and if instead of performing the following steps strictly through the API (which would enforce the `ctrl` settings) the system instead performs them directly.

Appendix A Future Features

`undo` (undoing previous commands)

It should be easy for interfaces to always provide an unobtrusive undo option in lieu of all confirmation dialogs. Note that we are already set up to arbitrarily undo IOUs since they are never deleted from the database and they indicate which ones replace which other ones. This audit trail can be followed using the `tran` call but it would be simpler to just call the undo command.

Appendix B Other Commands in Yootopia

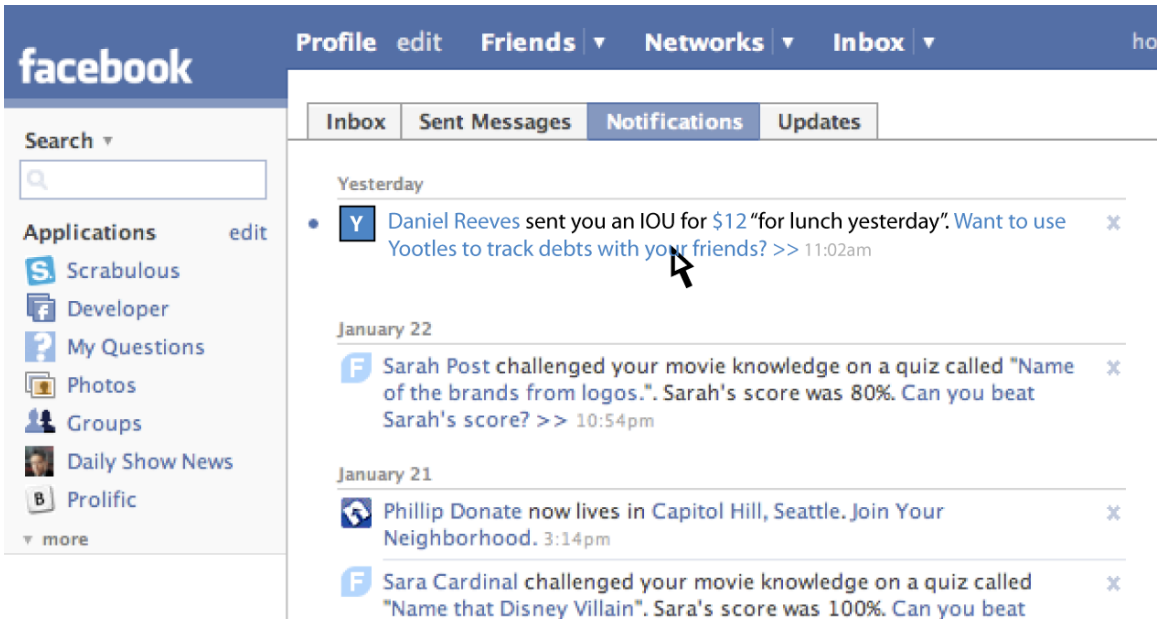
Some of the following commands are implemented for Yootopia Decisions and Predictions. All of them are registered on the open.4info.net SMS service.

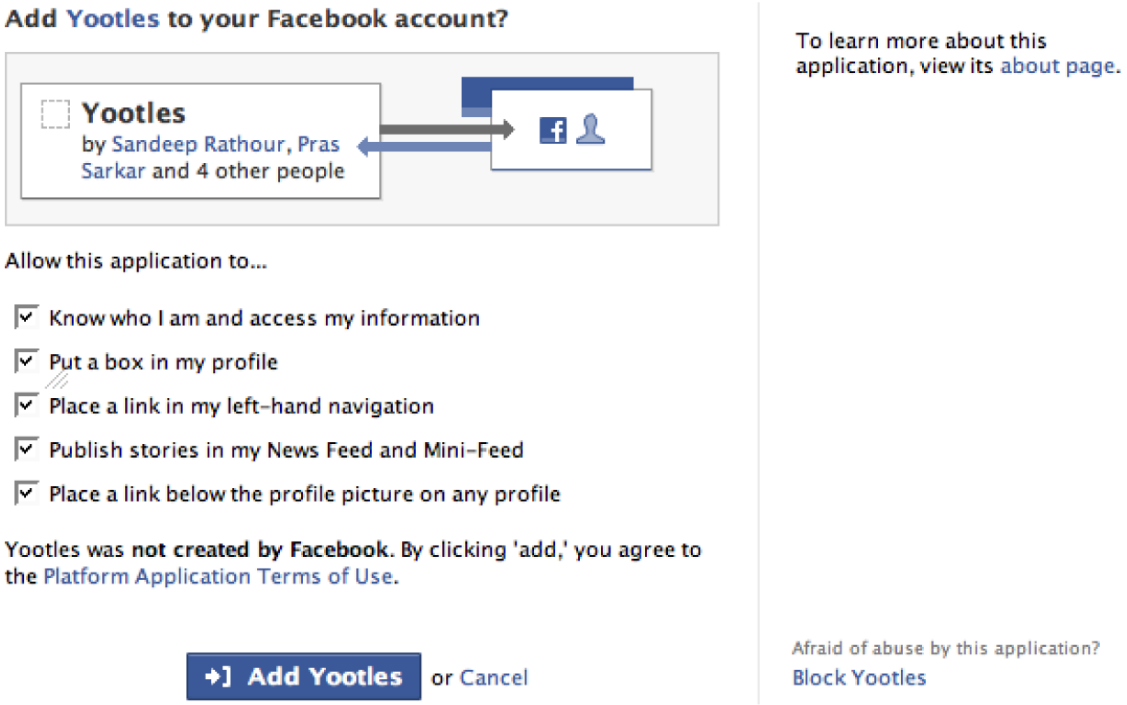
y0 y1 y2 y3 y4 y5 y6 y7
yad yadd
yalm
yant yante
yapp yapprove yapr
ybe
ybu ybuy
ycl ycle yclear
ycnt yco ycou ycount
ycur ycurr
ydir
ydon ydonate
yfav yfavor
ygrp ygroup
yho yhol yhold yholdings
yint yintr
yle yled yledg yledge yledger
ylog ylogin
ymer ymerge ymrg
ynew yold
yon yoff
yout
ypay
ypo ypok ypoke
yq yqu yque yquery
yre yref
yren yrename
yres yreset
yrm
ysel ysell
ysor ysorry ysry
ysto ystop
ysw yswi yswitch
ytha ythank ythanks ythx
ytr ytran
yun yund yundo
yunr yunreg
yvo yvoi yvoid
ywa ywag ywage ywager
ywhy
yab yabb yabbr yabr
yala yalarm
yand yor yxor ynot
yany
yas yask
ybi ybid
yby ybye
yclo yclose yclosed
ycp
yde ydel
ydis ydispute ydsp
ydow ydown
ygi ygiv ygive
yhe yhel yhelp yhlp
yid
yiou
yli ylis ylist yls
yma ymai ymail
ymo ymov ymv
ynom
yoo yoot yootles ytl
yowe
ype ypee ypeek
ypr yprd ypre ypred
yquo yquote
yreg
yrepu yrepudiate
yresolve
ys yse ysea ysearch
yset
yst ysta ystake ystk
ystats ysts ystt
yta ytal ytally
yti ytip
ytru ytrust
yunl yunlock
yup yupd yupdate
yvot yvote
ywai ywait
yy yye yyes yno yyay yay
yac yacc yaccept yacct
yali yalias
yannotate yano
yapi
yb yba ybal
ybin
ych
yclr ycls
ycr ycrd ycre ycred ycredit
yden ydeny
ydo ydid
yem yema yemail yeml
ygo
yhi yhis yhist
yinf yinfo
yj yk yl
ylo yloc ylock
ymch yme ymec ymech
yna ynam yname
ynote
yop yopen yopn
ypa ypau ypause
ypl yple ypledge
ypri yprice
yra yrat yrate
yrem yremind
yreq yrequest
yrf yrfq
ysee yseed
ysom ysome
ystatus
ysub
yth
ytog ytoggle
ytyp ytype
yunp yunpaid yunpay
yusr yuser
yvou yvouch yvch
ywh ywho
yyo yyou

Appendix C Facebook Application Mockups

C.1 Use case 1: Installing the Yootles application

A user could find the Yootles application through the Facebook application search, by seeing the widget on someone else's profile, or through a notification or invite in their messages or newsfeed.





The user is taken through the usual Facebook application install procedure, and then we take them straight to the IOU screen of the yootles widget. The application has a standardized way of automatically selecting a group and account name for their main account from the information available through Facebook so that there is no setup/registration within the yootles application beyond installing it.

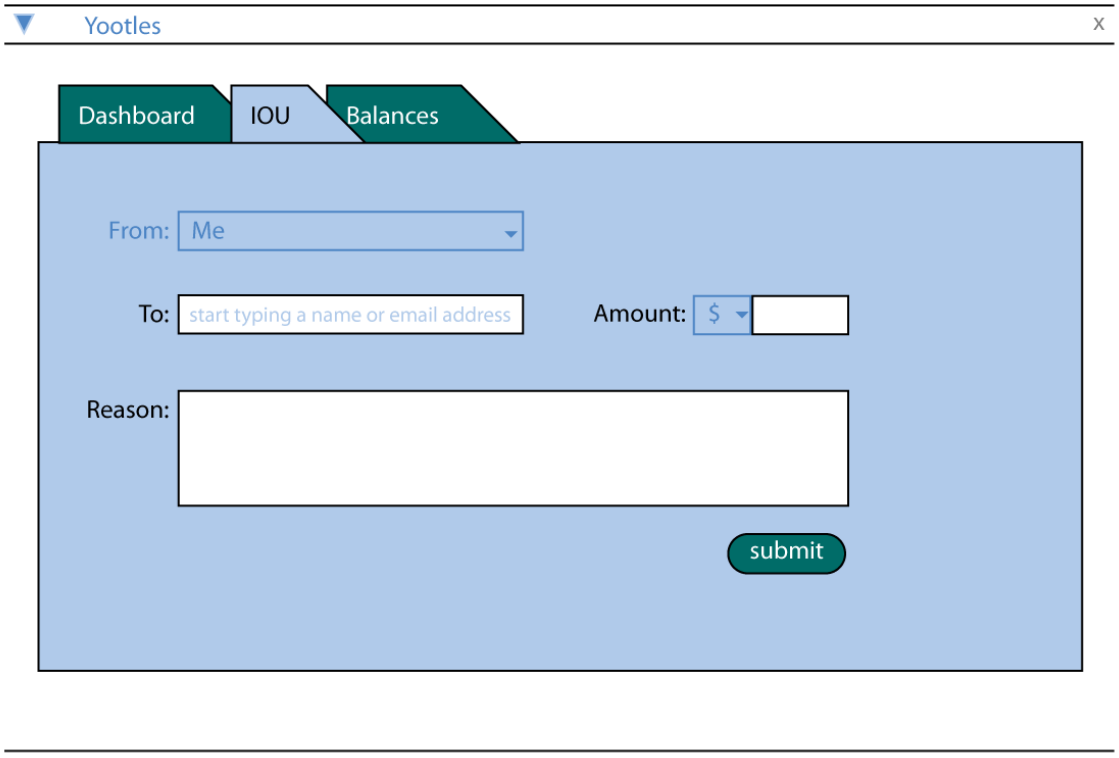
C.2 Use case 2.0: entering a simple IOU

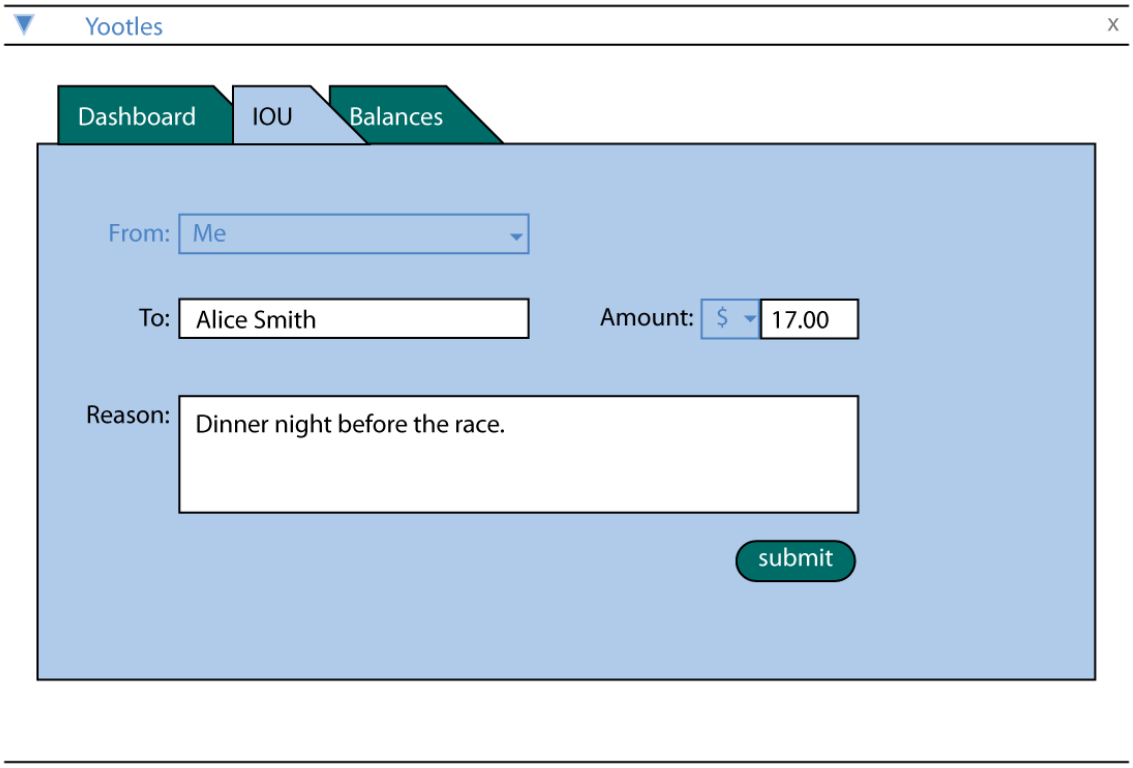
This is our most straightforward case of entering an IOU. Let's say you had dinner with Alice and she paid the bill. You enter an IOU for your portion.

The "from" field is filled in for you, defaulting to your main account.

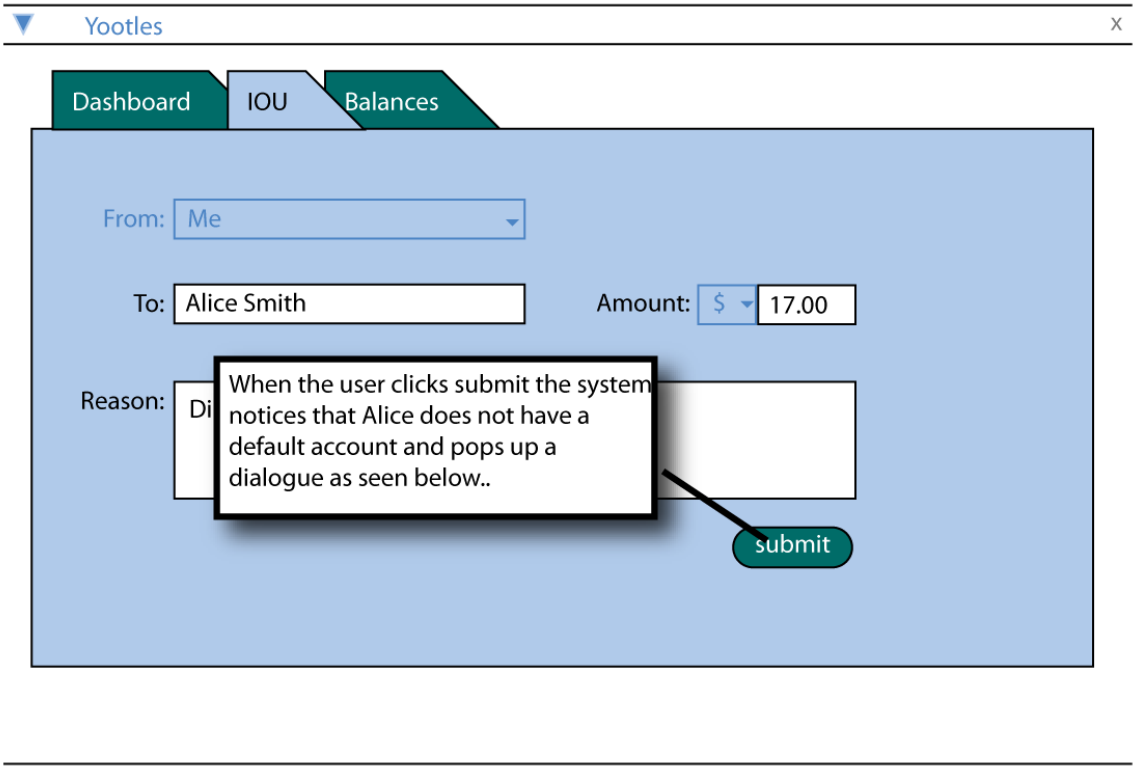
When you start typing in the "to" field your contact is auto-completed in Facebook fashion. Give an amount and a description string and submit.

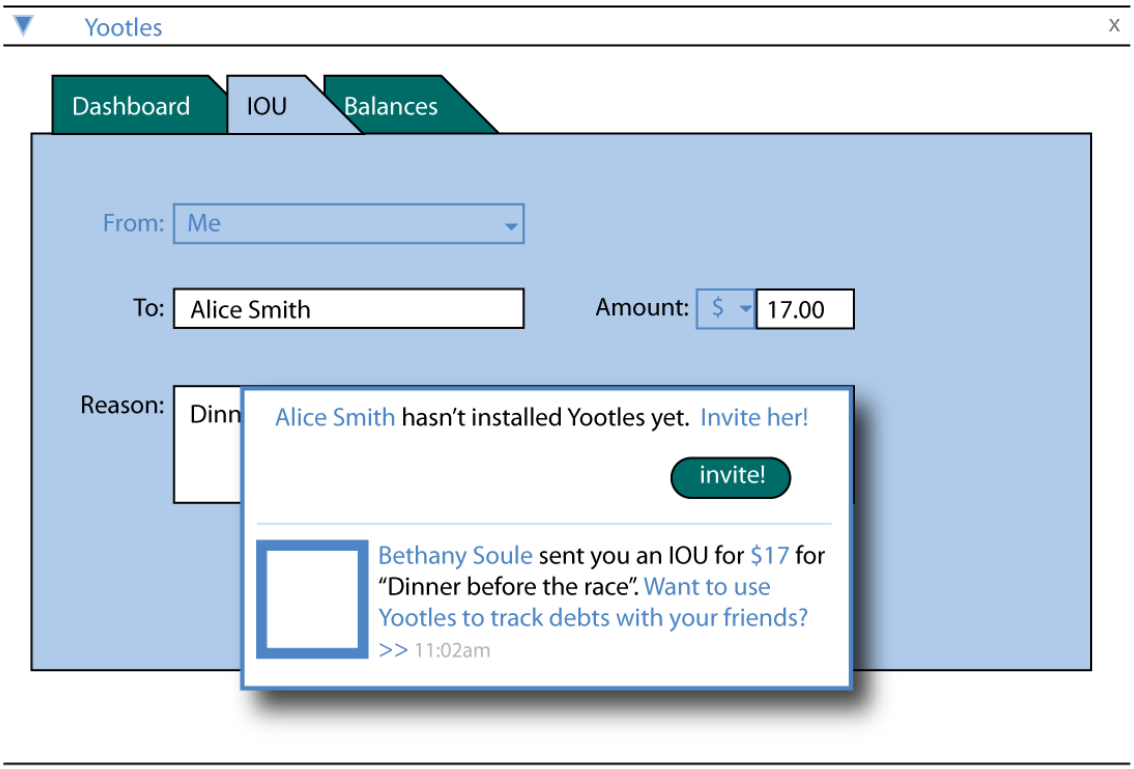
Notice also that the currency defaults to whatever the default currency for the "from" account is, but can be changed simply by clicking on it, or using the unobtrusive dropdown.




Yootles
x
Dashboard
IOU
Balances
From:
Me
To:
Alice Smith
Amount:
$
17.00
Reason:
Dinner night before the race.
submit

What if Alice hasn't installed the Yootles app yet? Then when the user clicks the submit button a small "invite" window will pop up. When they click "invite!" the invitation will be sent and the IOU issued.



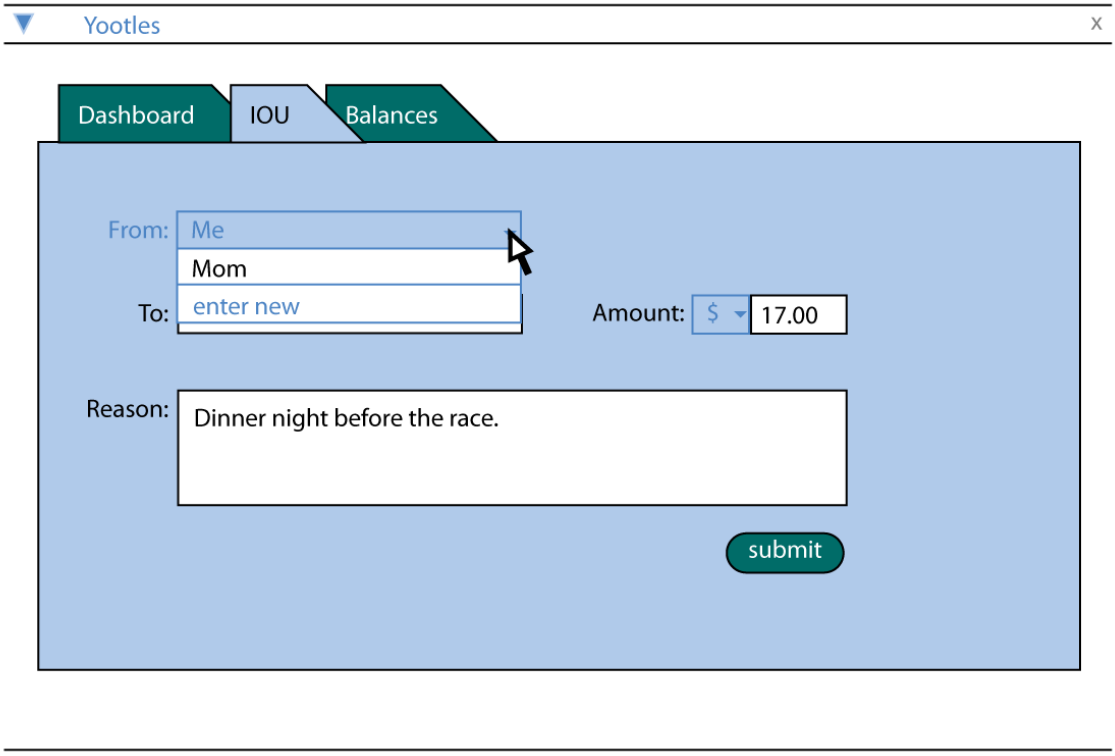


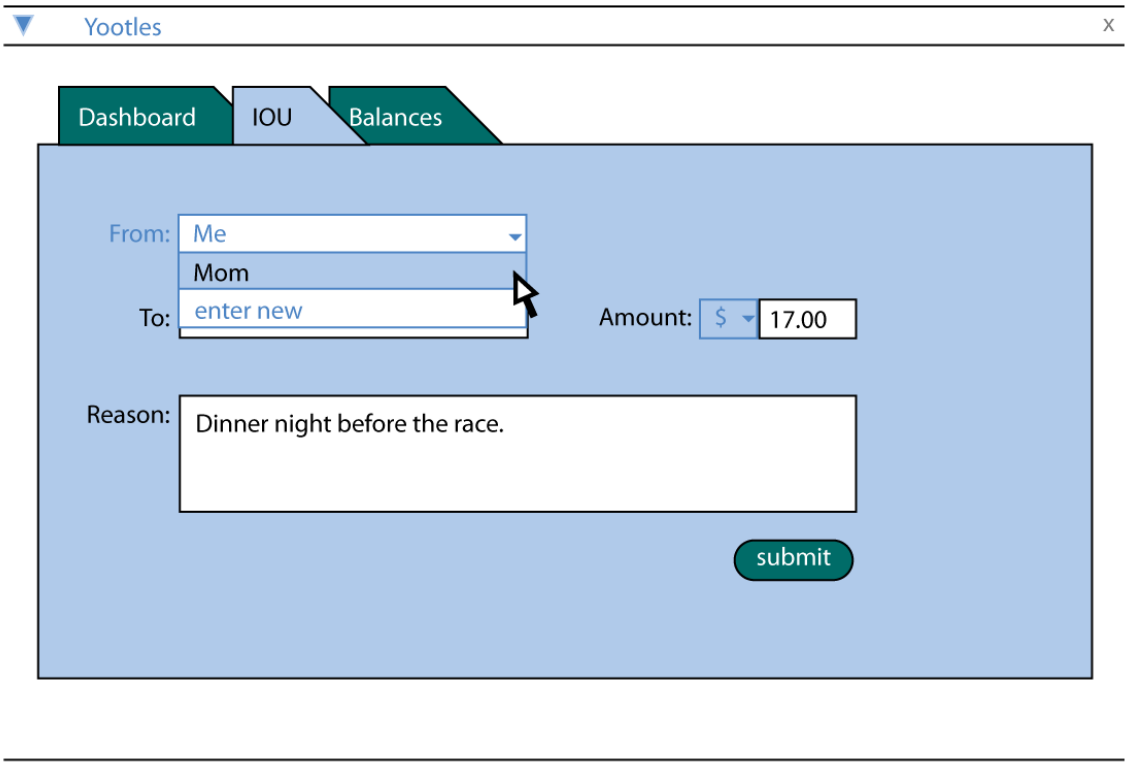
What if the person you want to issue the IOU to is not on Facebook? This would be nearly identical to issuing the IOU from "Dad" in the next use case (specifically see 2.1.2, 3).

C.3 Use case 2.1: entering an IOU for someone else

Say your mom also came out to dinner with you and Alice. You know your mom is never going to get on Facebook and enter her own balance with Alice, but you can enter the IOU for her. If you provided an email address for her when you created the account she will get email whenever you enter an IOU on her behalf with a summary of the transaction and perhaps some general balance info. If by some technical miracle your mom does someday show up to claim her account, she can review her past transactions, and modify them if

necessary. (Maybe she didn't understand the online banking thing and paid Alice back in person later on).

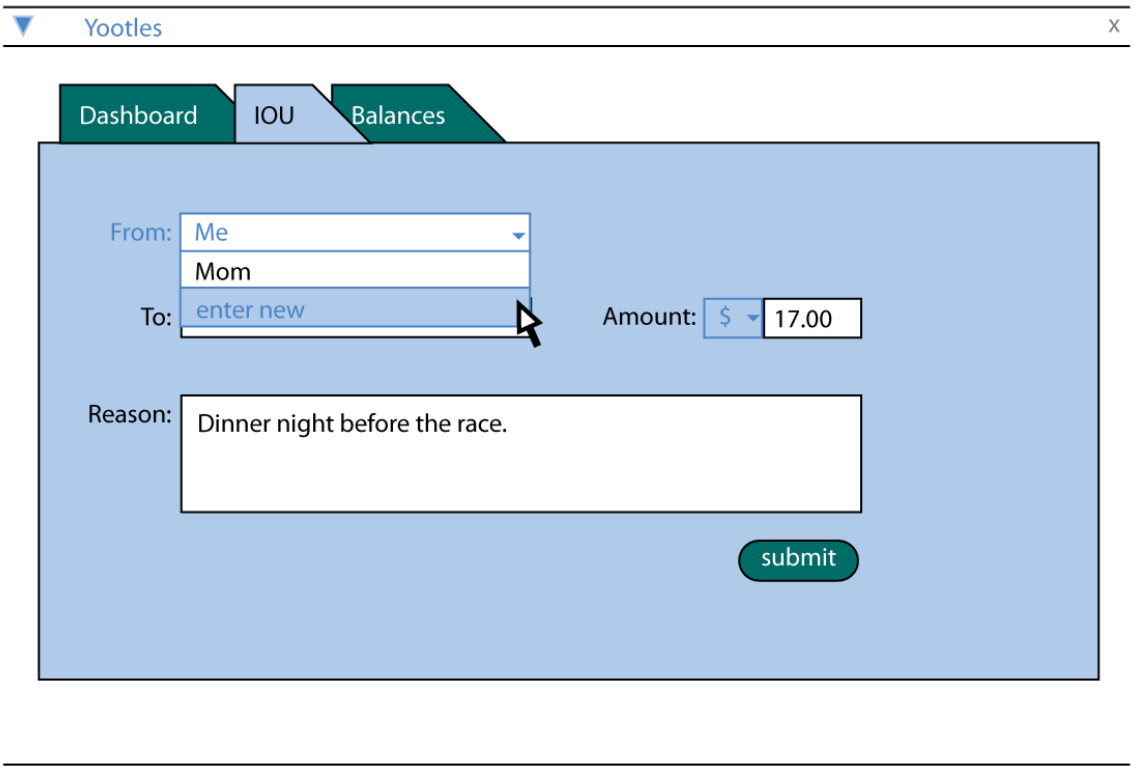


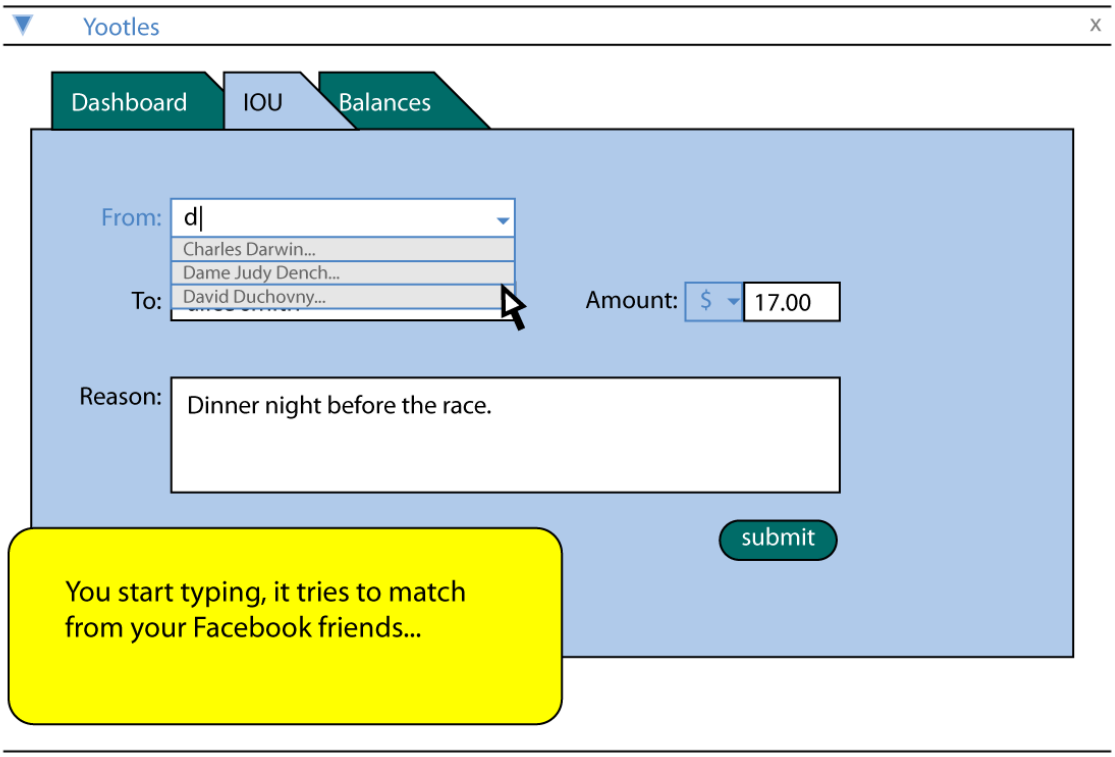


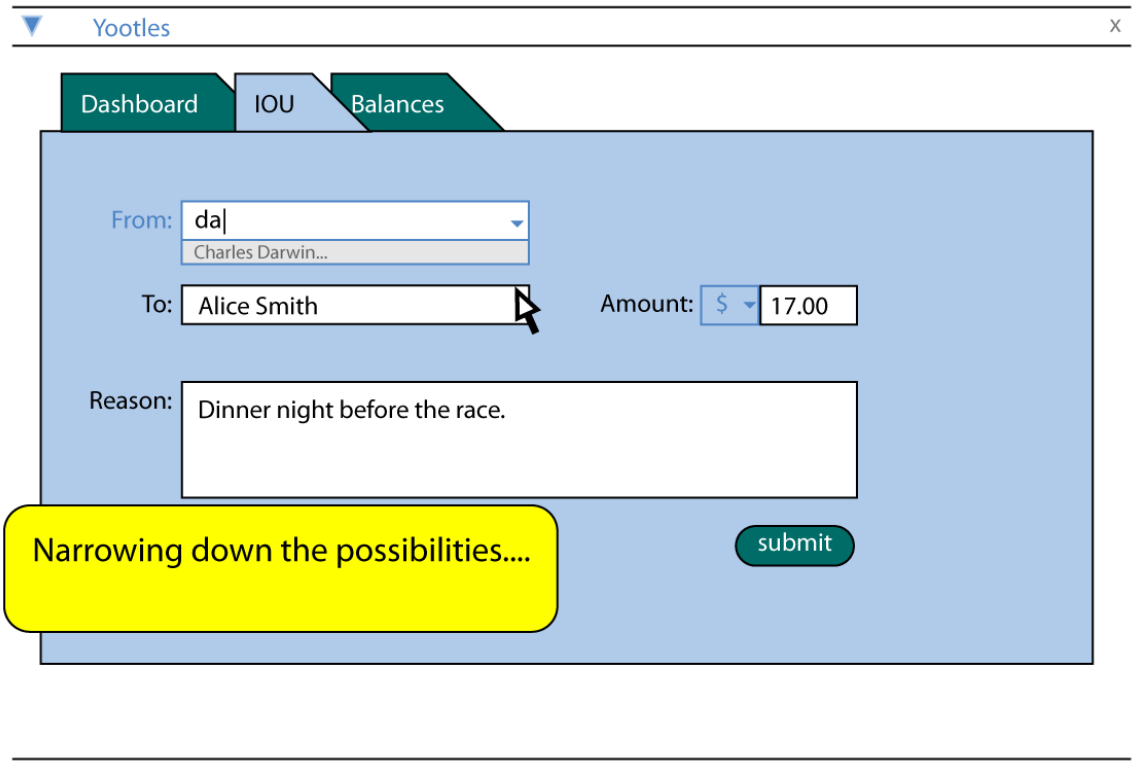
What if, instead, it was your Dad who was out to dinner with you, but you've never entered an IOU from him before, so his name doesn't show up in your list?

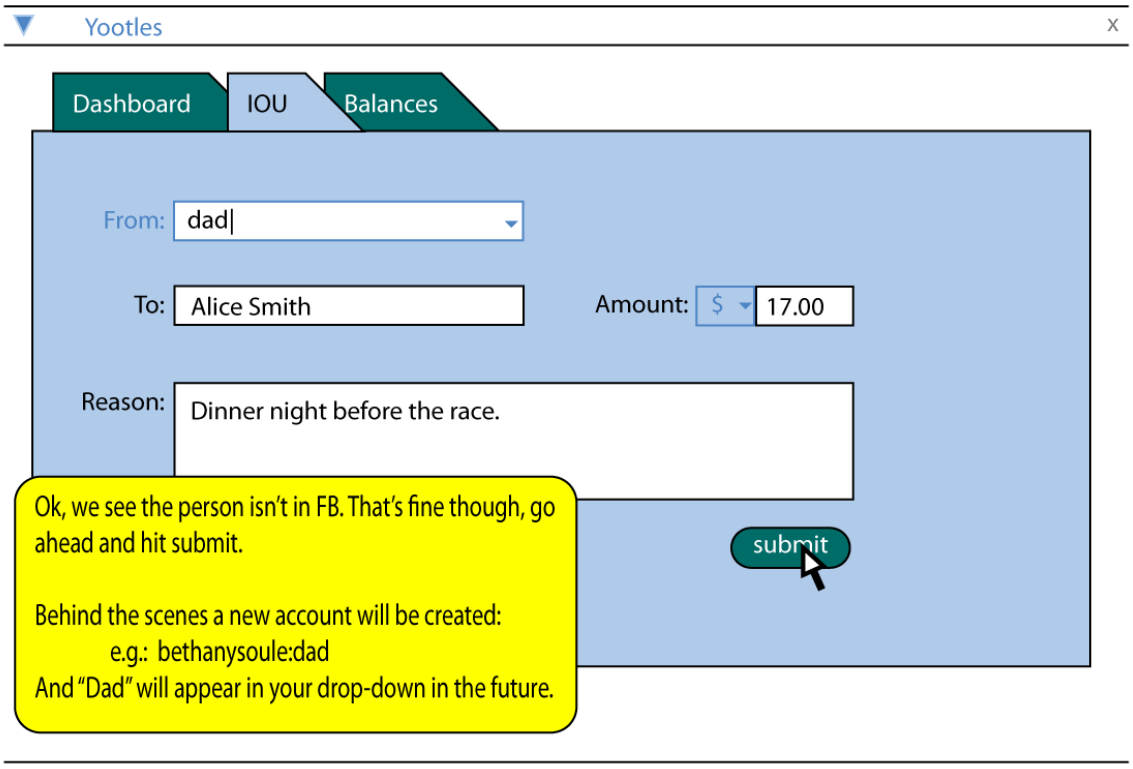
You select "enter new" and start typing. Facebook will try to complete for you from your friends list, but if the person is not on Facebook, that is OK too. A new account will

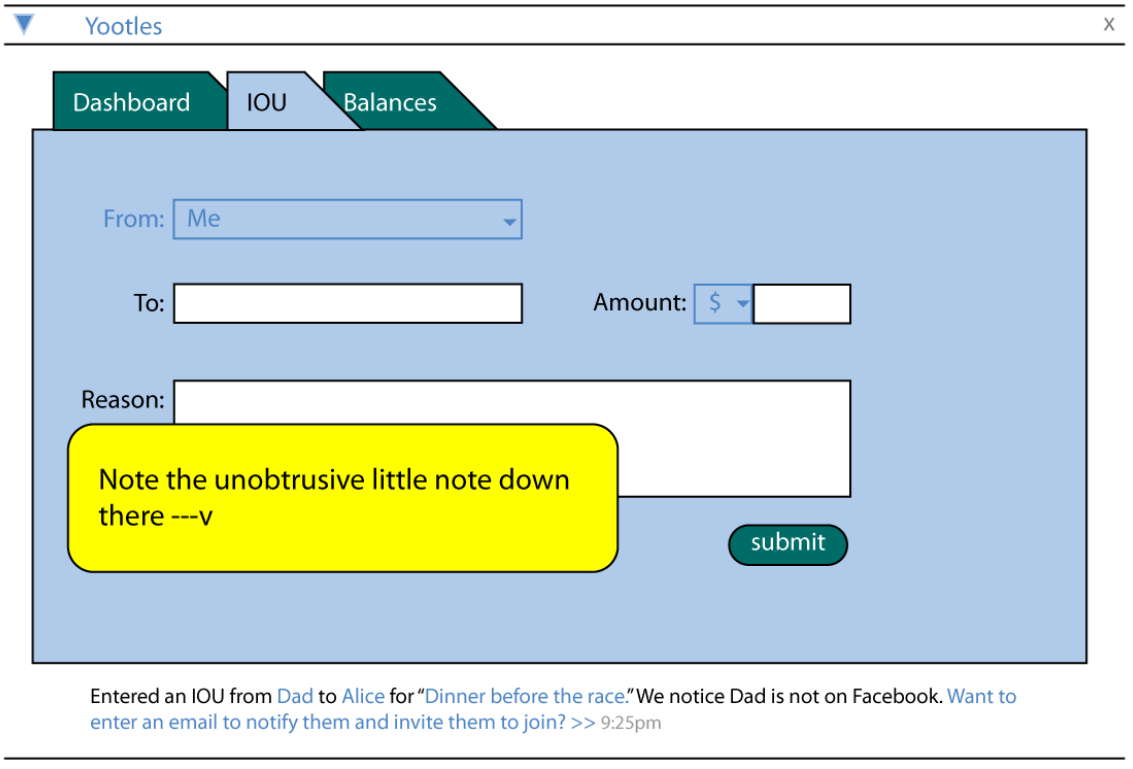
be created for them behind the scenes, and a little notification will show up underneath the widget after you have submitted (see the last illustration for this case).




Yootles
x
Dashboard
IOU
Balances
From:
d
Charles Darwin...
Dame Judy Dench...
David Duchovny...
To:
Amount:
$
17.00
Reason:
Dinner night before the race.
submit
You start typing, it tries to match from your Facebook friends...

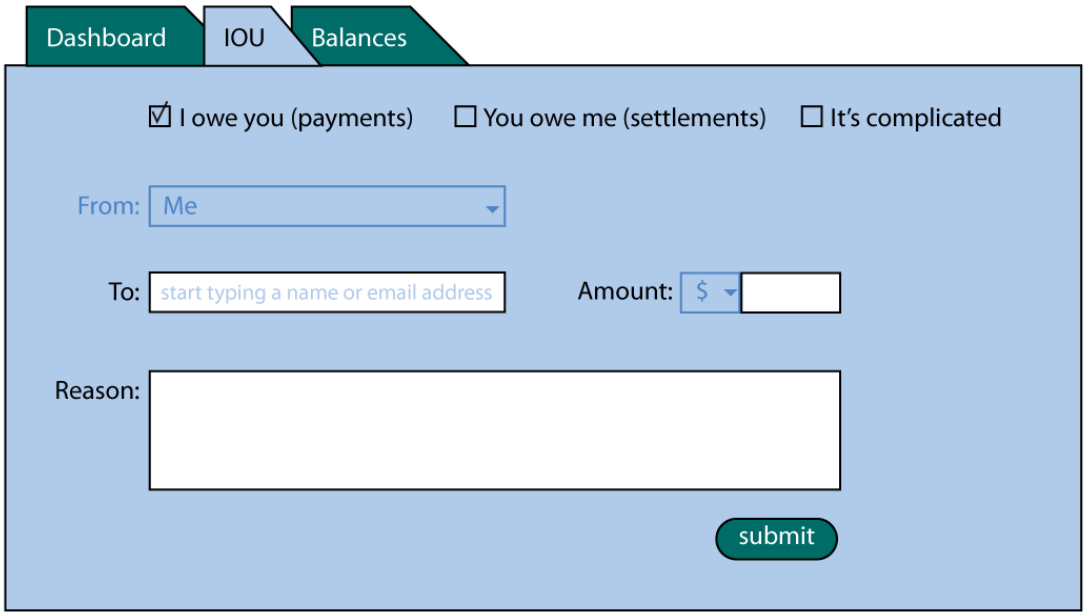

Yootles
x
Dashboard
IOU
Balances
From:
da
Charles Darwin...
To:
Alice Smith
Amount:
$
17.00
Reason:
Dinner night before the race.
submit
Narrowing down the possibilities....

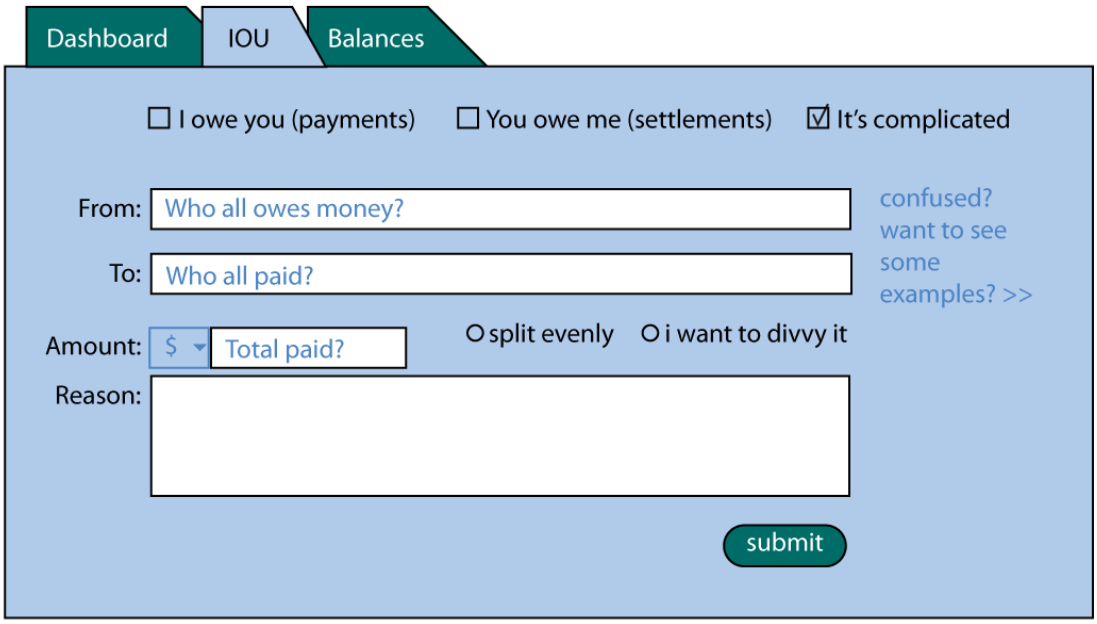

Yootles
x
Dashboard
IOU
Balances
From:
dad
To:
Alice Smith
Amount:
$
17.00
Reason:
Dinner night before the race.
submit
Ok, we see the person isn't in FB. That's fine though, go ahead and hit submit.
Behind the scenes a new account will be created:
e.g.: bethanysoule:dad
And "Dad" will appear in your drop-down in the future.

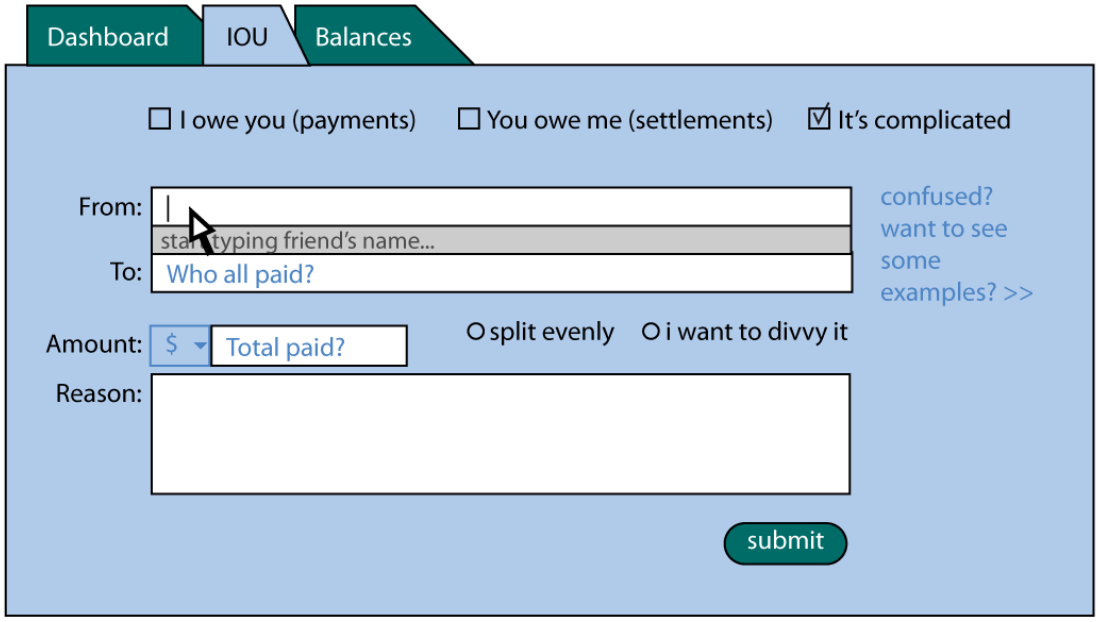

Yootles
x
Dashboard
IOU
Balances
From:
Me
To:
Amount:
$
Reason:
Note the unobtrusive little note down there ---v
submit
Entered an IOU from Dad to Alice for "Dinner before the race." We notice Dad is not on Facebook. Want to enter an email to notify them and invite them to join? >> 9:25pm

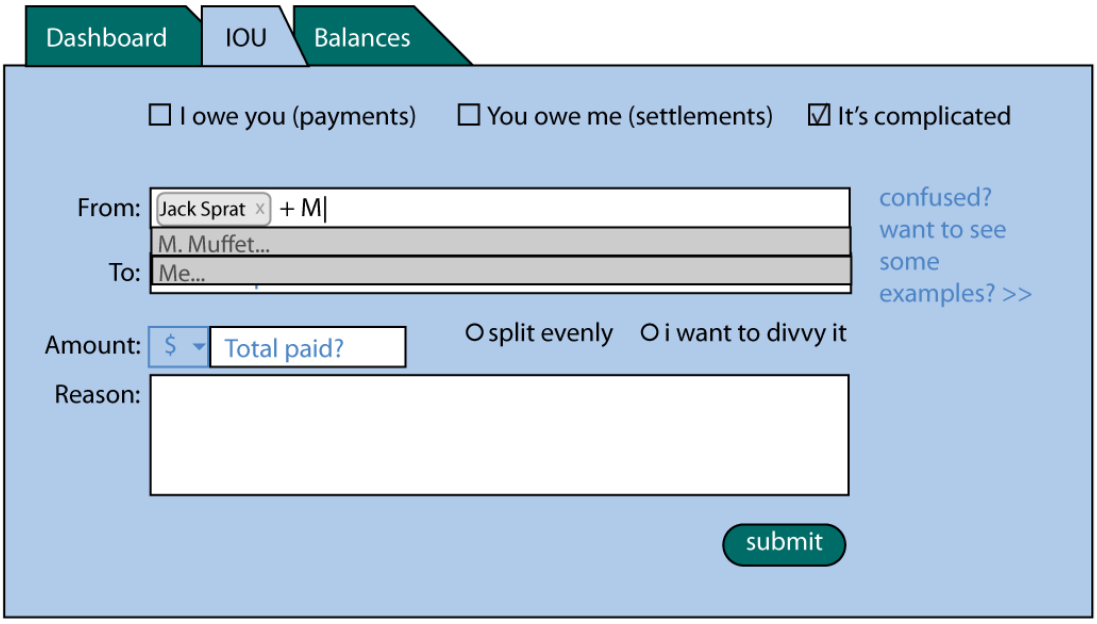
C.4 Use case 2.2: entering more complex transactions

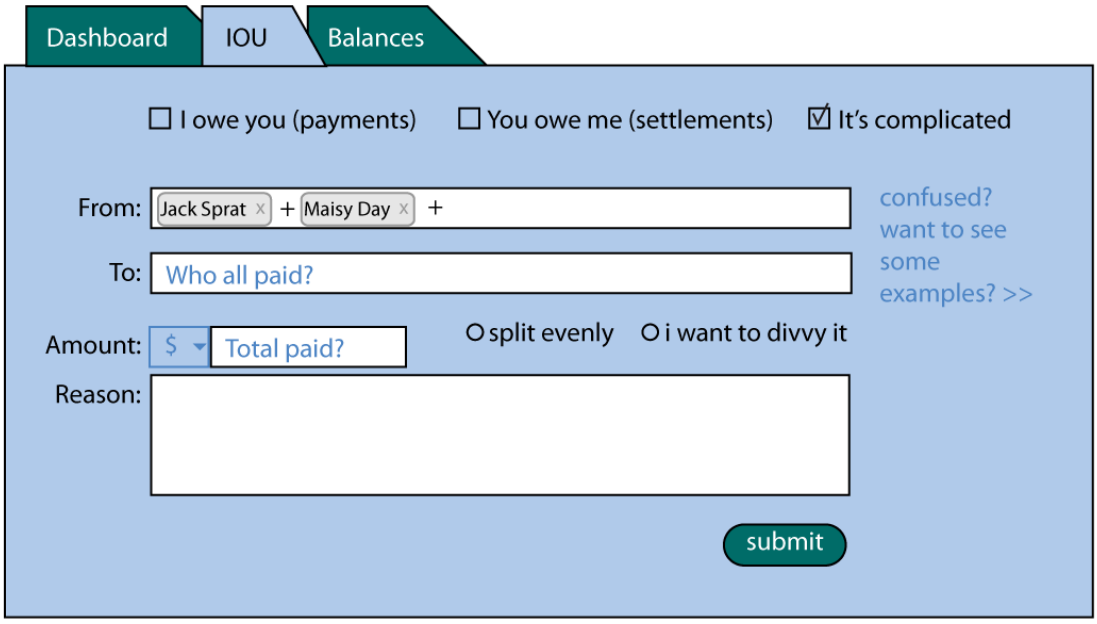
This is not a fully fleshed out example of how to lead a user through entering a more complex transaction, but hopefully it is a good enough beginning to give some ideas. The goal here is to provide a way to lead users through the process step-by-step, and have the widget construct the IOU in front of them, so that at the end of the process you would see how the complex IOU is constructed, and if you get it you could just enter it yourself next time (me+mo+larry+curly ... etc), but can always use the walkthrough if you need it.




Dashboard
IOU
Balances
I owe you (payments)
You owe me (settlements)
It's complicated
From:
Who all owes money?
To:
Who all paid?
confused?
want to see
some
examples? >>
Amount:
$
Total paid?
split evenly
i want to divvy it
Reason:
submit


Dashboard
IOU
Balances
I owe you (payments)
You owe me (settlements)
It's complicated
From:
start typing friend's name...
To:
Who all paid?
confused?
want to see
some
examples? >>
Amount:
$
Total paid?
split evenly
i want to divvy it
Reason:
submit

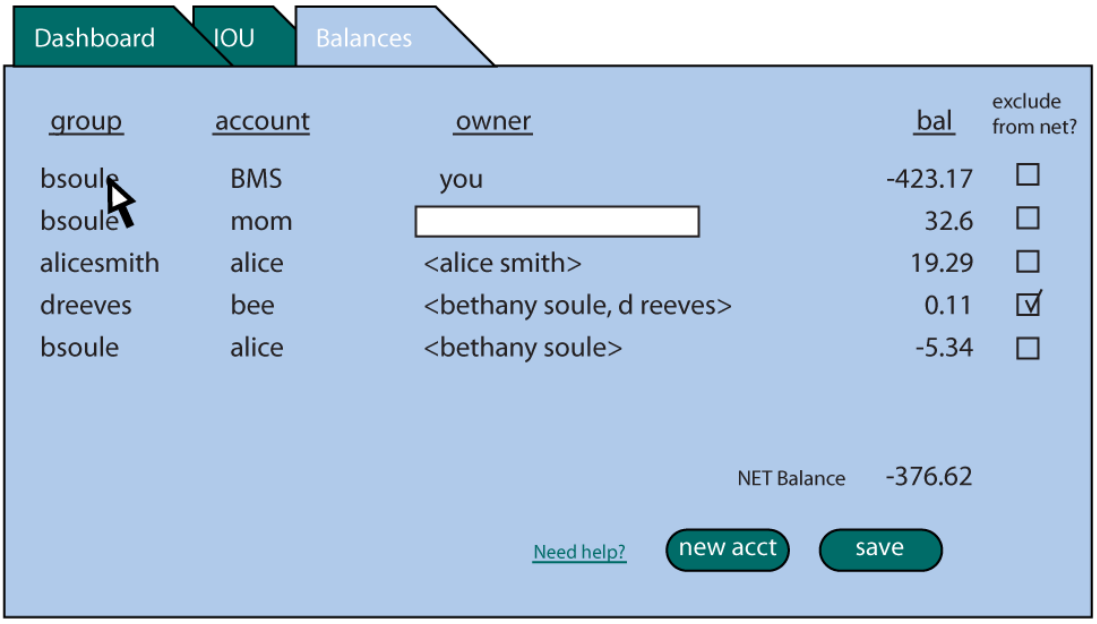


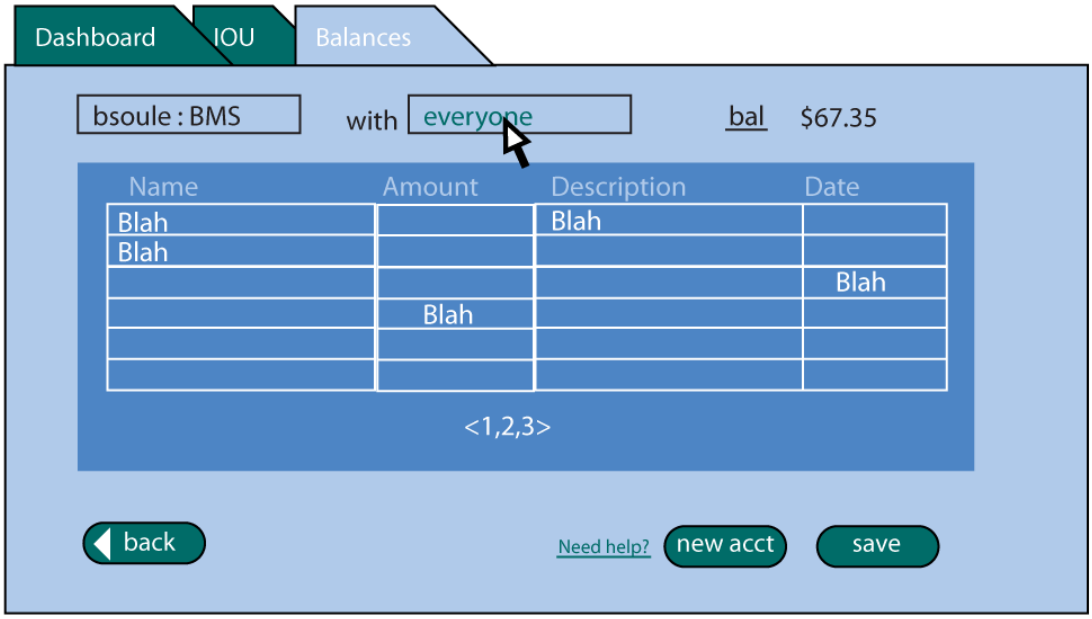


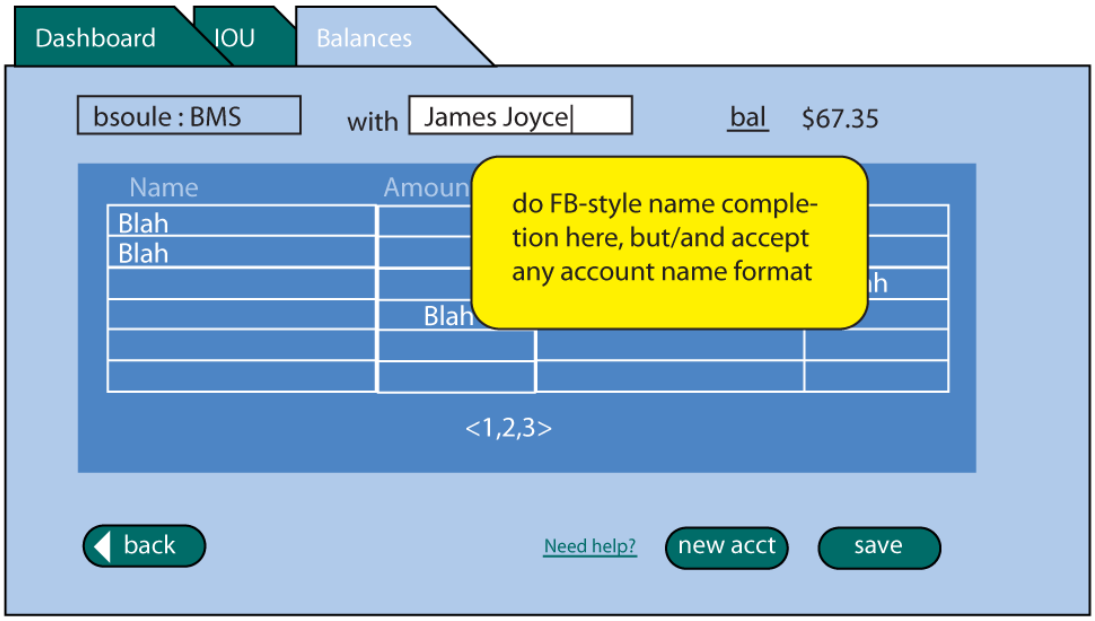
C.5 Use case 3: balances and managing your accounts

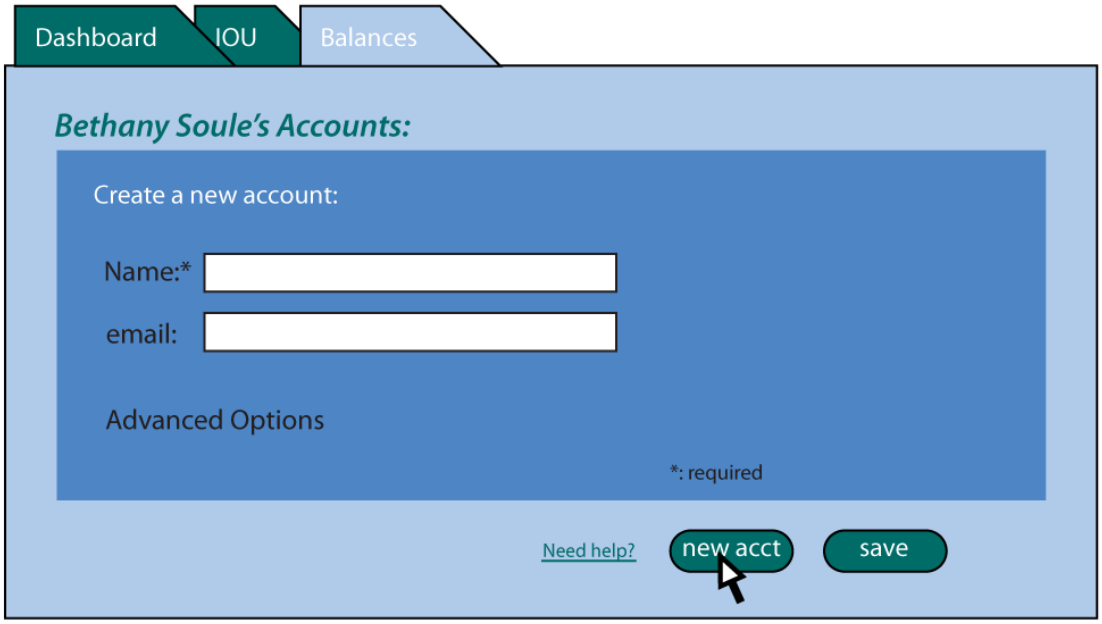
Here is a look at the balances page. This is where you would enter an email address for your mom if she wanted to get notifications about all the transactions you are entering that include her. You can see your net balance and look at the specific balances and transaction

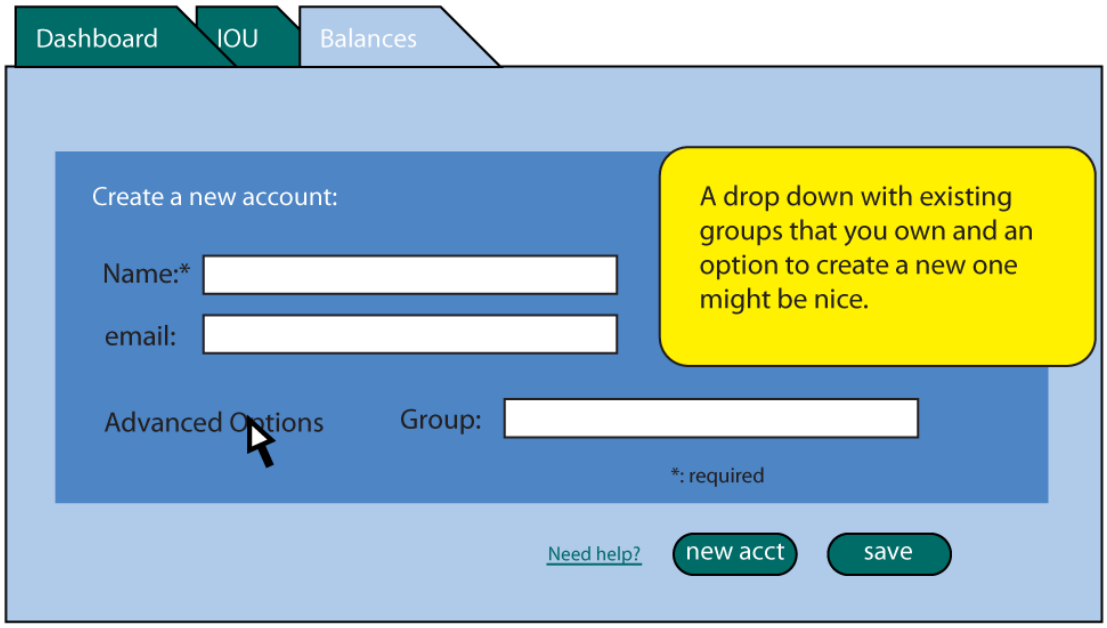
histories between pairs of accounts. You can also create new accounts here, and there is some information to introduce you to the idea of groups.

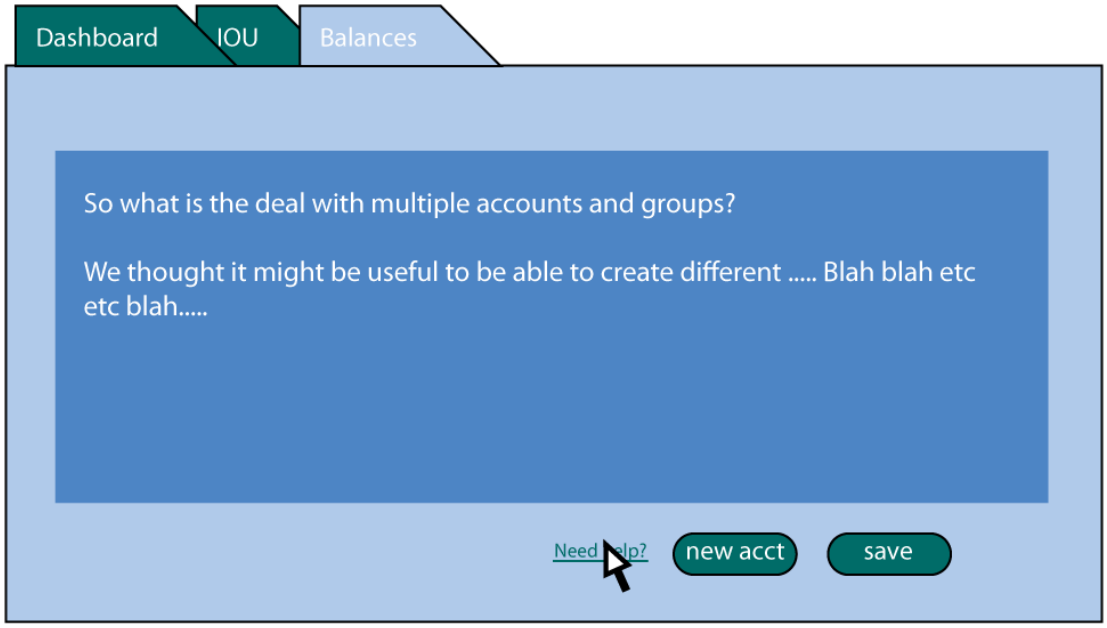





Dashboard
IOU
Balances
bsoule : BMS
with
James Joyce
bal
$67.35
Name
Amoun
Blah
Blah
Blah
do FB-style name comple-
tion here, but/and accept
any account name format
<1,2,2>
back
Need help?
new acct
save


Dashboard
IOU
Balances
Bethany Soule's Accounts:
Create a new account:
Name:*
email:
Advanced Options
*: required
Need help?
new acct
save


Dashboard
IOU
Balances
Create a new account:
Name:*
email:
Advanced Options
Group:
*: required
A drop down with existing groups that you own and an option to create a new one might be nice.
Need help?
new acct
save


Dashboard
IOU
Balances
So what is the deal with multiple accounts and groups?
We thought it might be useful to be able to create different ..... Blah blah etc etc blah.....
Need help?
new acct
save

Index

V

X

Y