TeamPlayer -

System Design Specification

**Table of Contents**

Architecture Process………………………….…………………………………………………..……………2-8

Architecture Design……………………………………………………………………………………………9-14

**I. Architecture Process**

**Risk assessment**

Here are some risks we are aware of as well as how we might fix or avoid them:

1. *Keeping up with our schedule and finishing all our core features in time*

This risk has a medium likelihood of occurring. Scheduling and timing is going to be an issue in any project but we have divided our project up well and planned a reasonable schedule to completing it. So if we stick to that we can greatly reduce the risk. The impact would be very high if it occurred, as falling behind schedule pushes everything else back, which can have a landslide effect and cause us to either have to cut features or not finish in time. We based our schedule on past experiences creating a web page and how complex we think each part will be. We will continue to adjust our schedule while we implement, as we may be pushed behind or find out other features we may need.

2. *Keeping backend synched with frontend*

The risk of here is not being able to synch the backend and frontend. That can either happen if we are not working on the same parts at the same time, or if the code is written in a way that we can’t connect the two. The risk of this happening is low because while we don’t always meet as a whole group, but at the same time we have a schedule we are all following so we should all be on the same page. And Karthik meets with both to make sure that everything is connected and going according to plan. If it does happen though, the impact will be quite high as we may have to go back and redo our code to integrate the parts because our app won’t work if the two parts don’t connect right. So to reduce the risk, we are making sure we have good team communication through weekly team meetings, talking through facebook and email, and Karthik checking both sides.

3. *Keep UI simple and useable, but still functional*

The risk is that we might go too far in one direction. We can potentially make the UI way too simple and not having enough functionality for users to use our application to manage their group easily. We also might make the application way too complex, during planning we often have new ideas, new features we could add, so we have to pay attention to the feature creep problem. We need to have a good balance between the two, which could be difficult. Especially because we are biased because this is our product, it is easy for us to think a UI is simple to use when it actually isn’t. The impact of this is medium, because the app will still have all of its features available, but it will just look different confusing and uninviting. Depending on how simple or how complex it is, it can push some users away from using our app. To try and reduce this risk, it is important for us to keep having customer meetings because that is a great way to get an outside opinion on our UI. Once we have a working UI, we can also use hallway usability tests to see what new users think of our UI.

4. *Have everyone work close enough to help each other, but on separate tasks for*

*efficiency*

The risk is that we may not balance between the two well. For instance, if we always work together as a group, where all seven of us are sitting together, we might not get as much done because we will end up all working on the same part of our product. On the other hand, if we always work individually and just merge everyone’s work, we might not be able to understand each other’s work and merging becomes difficult. If we don’t understand other’s work, it is impossible to understand the product as a whole. It also increase the difficulty of debugging other’s code. The likelihood of this occurring is low, as we have separated into smaller groups to do group work, and also assign task for everyone to work on individually. On the other hand, we also have regular group meeting for everyone to communicate with the whole group. In other words, we have been using an approach that deals with this problem. The impact of this risk is medium, especially later in the development process since as the project grow, it is harder to understand what other did in the past. To detect this, we will have weekly meetings where everyone will talk briefly about what they think others have done. If someone is clearly not understanding the group progress, we need to get everyone up to date by showing each other what we have done. Also, we need to adjust our meetings to make communication more clear and efficient.

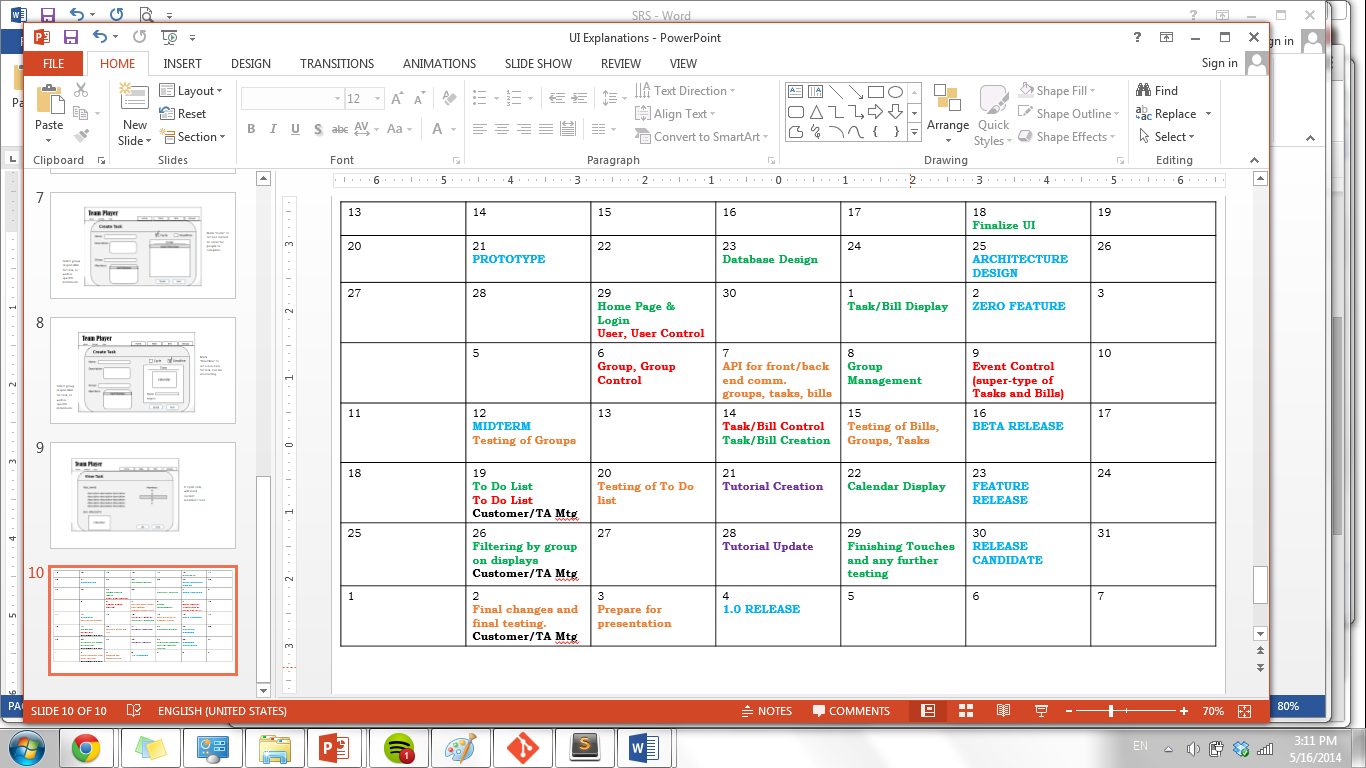
5. *Create variety of options for events to be simple but fit all needs of user*

This is a pretty big risk for us because we are trying to add enough features to meet the needs of our users, such as tasks or bills that may not always have a date. Or repeating/cycle tasks like taking out the trash or getting milk which may not always have a set date. But depending on how we implement these features, it can be extremely cluttered and difficult for the user to use. The create event feature would be extremely complex with too many options and unfriendly to users. So the impact is high because this is one of our most important features that differentiates our app from other similar ones. So we need to add these features in a way that is easy to use and understandable to the user, but also helpful. To reduce the risk here we need to do a lot of user testing through customer meetings and hallway usability tests specifically for this portion of our app.

Our risks are pretty similar to the risks we had from the SRS, except now we have some more specific risks because we know more about our app. Timing and integration between frontend and backend are always going to be risks that can have a huge impact on the project throughout the whole entire process. But now that we have spent more time working on the UI and making design decisions according to feedback from customers, we better understand what will be difficult and is a high risk for the project. We know that the events cycling will be difficult, but necessary, and also understand that creating a simple, usable UI that is pleasing to users is difficult.

**Project Schedule**

Similar to the schedule in our Software Requirements Specification, here is another view of how we plan to implement our product.



It is helpful that we are able to separate our features into different modules: bills, tasks, groups. Although the functions are relatively related to each other, we can work on classes and views independently. The calendar above is taken from our SRS, and from TA comments as well as a bit more understanding about our process, we have rearranged some of the deadlines so that our workflow is more practical and efficient. In the SRS, we had put bill and task creation before making the display, but it makes sense that we should know what we want to display before we understand what a user should input to create events. Also, we had realized that creating groups are important to creating tasks and bills. So we need to do that in parallel with even creation.

In blue are CSE403 deadlines, green are front end view deadlines, and in red are backend classes deadlines. Additionally, in purple are deadlines for tutorials – first to create, then to see whether our customers know how to use it, and in black are customer and TA meetings to make sure our app is usable. Our features could actually be worked on independently, however getting front end to synchronize with backend will be a challenge. This is why we have different feature focused weeks.

When we can run features:

- *log in*: By April 30, users will be able to create an account and get to home page.

- *bill/task creation*: This is also depending on group management because bills and

tasks are group central, meaning can belong to a group and include groups of people. This is why group classes must be created along with bill and task classes. We will be able to run this feature by May 10.

- *group management*: By May 14, users will be able to view and edit groups.

- *calendar display*: Given all of the events to display, we will need to be able to filter

them by May 22.

**Team structure**

*Team Structure*:

Project Manager and front-end-back-end integrator: Karthik

Front End developers and testers: Fung, Linsen, Tiffany

Back End developers and testers: Keith, Micaela, Panji

*Tasks*:

Our priority is to complete the login feature, to allow the foundation to build the user base that would interact with other features such as group management, creating event,  task, bills. So in the first week we all need to focus on login, and since we are new to the framework working together in a small feature will be a good warm up before we each of us are responsible for a feature specific to our modules. We can then divide features up amongst us: groups, tasks, bills, where one front end developer and one back end developer is focused on a module. Though, since some modules have functions very similar to each other, we will be able to work together and help each other out.

*Meetings*:

We have meetings with our entire on Monday to plan the coming week’s goal more specifically, and to think about our individual and team status report. Then, on Tuesdays we meet with the entire group again to code together side by side, clearing up any misconceptions and helping each other out. For the rest of the week, we have separate frontend and backend group meetings later in the week to plan work on specific modules and divide individual tasks.

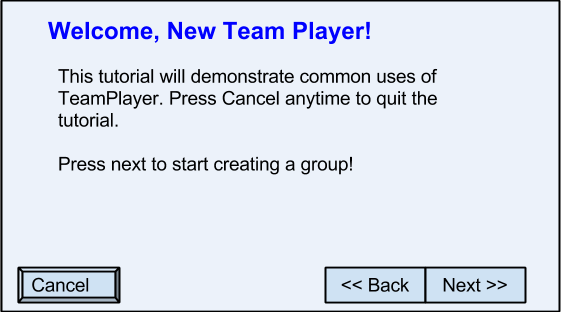
We have a google group and facebook chat to communicate with each other constantly. We also post meeting notes on our wiki so that we can remember things discussed. We often email meeting summaries to make sure we are all on the same page after meeting.

**Test plan**

* Unit test strategy
  + The unit tests will test most (not all as small private functions may be tested as part of other functions) functions in a black box and glass box way.
  + Ruby on Rails provides an integrated unit testing system so making tests will not be very complicated. Tests will be written by the developer of the function. We will not write major functions without testing them thoroughly. Here is a good reference tutorial and style guide to follow: <http://guides.rubyonrails.org/testing.html> for specifics of how we would unit test controllers in Ruby.
  + AngularJS (frontend) also provides a unit testing system. It is a little more complicated because it deals with simulating certain events, but we plan to follow a similar strategy. The major controller functions and model functions will be tested in parallel with developing them. Here is a good reference tutorial we will use: <https://docs.angularjs.org/guide/unit-testing> for the specifics on how testing works in Angular.
  + We plan to create tests while developing a class so testing should be done once a function is done. We will require that before pushing code to the master branch, all of their test code must pass and seen by at least one team member. No automation system seems necessary at the moment--we can just run the tests before pushing.
  + In general, we may not have the backend implemented at the same time as the frontend (and vice versa), so we will likely hardcode dummy JSON objects on both ends to use for testing, and actually connect them during system testing.
* System test strategy
  + The system test will test the integration of composite classes and the connection between the backend and frontend. While unit tests on each end used dummy JSON values, now we will actually allow the frontend and backend to communicate over the wire. These tests will mainly check that this communication works correctly.
  + We will mainly do systems testing on the frontend. We can use similar (or even the same) tests from the unit tests, and now ensure that sending/receiving data on the frontend and backend work correctly.
  + We will do system tests after every major component (e.g. viewing and creating events) is done both on the frontend and backend. Again, we are not going to use automation testing.
* Usability test strategy
  + We will mainly focus on hallway usability testing, because we have a limited amount of time for the project. Our strategy will be to identify major components of the UI we need feedback (as we are designing them), just like the paper prototype, and try to ask specific questions of the random people we find.
  + This will mostly give us feedback specifically on whether is a component is \*simple and intuitive\*, but will not give us feedback on whether higher level ideas components like “automatically splitting bills evenly” is useful or will be used a lot.
  + This will be done while developing. We plan to develop the product in a sequential way--implement the high level features one at a time together. Then, when a use case is completely finished (or at least the UI portion), we will try to grab people to try the system to see if it is usable.
* We will be using GitHub’s issue tracker, where any issue not merged yet will be dealt with in pull request but any bug found after merging will be posted on the tracker. We will essentially follow the guidelines from the Joel’s Test--find bugs early (testing), fix bugs early (don’t forget about them), and formally keep track of bugs (Git).

**Documentation Plan**

With the web application, we plan to deliver guides that helps user to get familiar with our product. For first time users, there will be a pop up tutorial that walks through the common use cases. The first pop up would be like this:



Each time the user hits “Next”, it goes to the next step with instructions on performing a use case. For instance, to guide the user to create a group there will be steps like, (1) click on “Create group”, (2) input Group Name and description, (3) Add members, (4) click “Create”.

For non-first time users, or users who closed the tutorial, will have a chance to see the tutorial again by going to the “help” page and click on “see tutorial”. Moreover, each page will have a small “?” icon in the corner where information about the page will pop up; there will be arrows pointing at page elements with labels that explains what that element is. Also, users can go to the “help” page and navigate to the help page for each individual pages. These individual pages will have the option to expand and show more details and spec about the functionality of the page, these details will not be shown at the initial tutorial popup to avoid overwhelming users.

**Coding style guidelines**

Here are our guidelines for using HTML/CSS, AngularJS/JavaScript, and Ruby On Rails:

* HTML/CSS
  + We will be using the google HTML/CSS guidelines,

<https://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml>.

We will look over CSS best-practices which optimize browser rendering in this

page here: <https://developers.google.com/speed/docs/best-practices/rendering>

For HTML and CSS there are validators like <http://validator.w3.org/>. Stylewise, there are options like CSS Lint <http://csslint.net/>.

* JavaScript and AngularJS
  + We again will be using the google style guide as well as the e basic  JS

guide <http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml> In addition to that, AngularJS is always expanded on the JS guide, <https://google-styleguide.googlecode.com/svn/trunk/angularjs-google-style.html> For more coding checkers, we can use the CSE 154 web programming class’s JS Lint <https://webster.cs.washington.edu/jslint/> as well as code review by our team members.

* Ruby on rails
  + Unfortunately this doesn’t have a style guide from google or other major source, but we do have style guide for rails and for ruby. We will be using these two from github, <https://github.com/bbatsov/ruby-style-guide>. We also have a ruby static code analyzer from the same person which can enforce these styles <https://github.com/bbatsov/rubocop>. Overall we have to code review each other’s code. Since we are all pretty new the these new frameworks so when we look and understand each other’s code we also learn more about the language itself too.

**II. Architecture Design**

**Data Storage**

In our system, all the data is provided by users. Data that the system stores include each user’s account information, and information of tasks, bills, and group created by users. The database will store the user’s first name, last name, username, password, and a user ID. For tasks and bills, the database will store the users involved, the user who created the task, if it’s cycled, which dates it’s repeated on, the date due, the date created, a title, a description, and if the event has been completed. Bills will additionally store the total of the bill and the costs for each individual involved. The database will store group’s users, group ID, date created, description and title. The SQL schema of the data is provided below:

CREATE TABLE users (

id MEDIUMINT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

username CHAR(50) NOT NULL,

fname CHAR(50),

lname CHAR(50),

);

CREATE TABLE groups (

id MEDIUMINT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

cuid MEDIUMINT NOT NULL,

name CHAR(50),

description CHAR(255),

gInvite bit,

self bit,

FOREIGN KEY (cuid) REFERENCES users(id)

);

CREATE TABLE ug

(uid int,

gid int,

FOREIGN KEY (uid) REFERENCES users(id),

FOREIGN KEY (gid) REFERENCES groups(id)

);

CREATE TABLE tasks (

id MEDIUMINT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

gid MEDIUMINT NOT NULL,

cuid MEDIUMINT NOT NULL,

title CHAR(50),

description CHAR(255),

date\_created date,

date\_due date,

date\_finish date,

g\_modifiable bit,

FOREIGN KEY (cuid) REFERENCES users(id),

FOREIGN KEY (gid) REFERENCES groups(id)

);

CREATE TABLE bills (

id MEDIUMINT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

gid MEDIUMINT NOT NULL,

cuid MEDIUMINT NOT NULL,

title CHAR(50),

description CHAR(255),

total MEDIUMINT,

date\_created date,

date\_due date,

split bit,

FOREIGN KEY (cuid) REFERENCES users(id),

FOREIGN KEY (gid) REFERENCES groups(id)

);

CREATE TABLE specevents (

id MEDIUMINT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

gid MEDIUMINT NOT NULL,

cuid MEDIUMINT NOT NULL,

title CHAR(50),

description CHAR(255),

repeats CHAR(8),

cycle BOOLEAN,

date\_start date,

date\_created date,

last\_finish date,

last\_id MEDIUMINT,

last\_modified date,

FOREIGN KEY (cuid) REFERENCES users(id),

FOREIGN KEY (gid) REFERENCES groups(id),

FOREIGN KEY (last\_id) REFERENCES users(id)

);

CREATE TABLE tasksm

(tid int,

uid int,

FOREIGN KEY (uid) REFERENCES users(id),

FOREIGN KEY (tid) REFERENCES tasks(id)

);

CREATE TABLE billsm

(bid int,

uid int,

FOREIGN KEY (uid) REFERENCES users(id),

FOREIGN KEY (bid) REFERENCES bills(id)

);

CREATE TABLE speceventsm

(sid int,

uid int,

FOREIGN KEY (uid) REFERENCES users(id),

FOREIGN KEY (sid) REFERENCES specevents(id)

);

The user, bill, task, and group information is sent from the website to the database by a JSON query. After validating the information, such as whether the username has been taken, whether the sub prices add up, it is stored in a SQL database. When this information is queried by a user on the front end, it is stored in a model (user model, group model, bill model, or task model). These models store the same information as the backend models but can be easily accessed by the UI and user.

**Design Decisions**

A major design decision was around Ruby on Rails. We had to decide between using the full power of Rails—frontend to backend, or, a mix of Rails (backend) and AngularJS on the frontend. We eventually decided on the mixed method because AngularJS has many features that simplify our UI implementation. Here are the main pros and cons of our chosen method:

Pros:

* We can better split up the frontend/backend groups, so each can work independently without worrying about the implementation of the other group. This would have been possible in Rails, but we decided it would be easier this way.
* AngularJS allows us to have a MVC on the client side—so as the model updates, the view will instantly be updated.
* We still could have used Angular even with using all of Rails, but it would limit the functionality that we could have used from Angular.
* Doing a lot more work on the frontend is better because more of us know JavaScript than Ruby. Either way, we felt JavaScript and Angular would be \*much\* easier to learn than Ruby and Rails.

Cons:

* We have two frameworks to learn rather than just one. Since most are only dealing with frontend or backend, we each probably would only need to worry about learning one.
* We probably increase the amount of code we have to write, because we now have two sets of models—the databases Events, Users, and Groups, and the parallel AngularJS models. We also have two sets of controllers for Events, Users, and Groups. This will not have a lot of redundant code, because they have very different responsibilities, but it will make the system larger than it would have been otherwise.

Another design decision where we considered an alternative was with our todo’s vs tasks. We spent a long time trying to figure out how we were going to implement these, and were seriously considering not implementing them as an option. It came down to whether or not we believed it was a serious need for people when creating a task and if we thought people would actually use it. We brainstormed tasks that we might put on our app, like taking out the trash or buying milk. We decided that these really didn’t need a date, and would be useful to create a new type of task for these.

Pros:

* They would be much simpler for the user, you just check it off when you finish and then it becomes someone else’s turn. We could see this being useful so that people don’t have to worry about dates, and just handle the task when needed.

Cons:

* It adds a new option to tasks that can be confusing, and we weren’t sure how to fit into the UI both in creating tasks and in the home page.

Finally, we discussed how to synchronize the backend database to the frontend model. This discussion is directly connected to the one above with Rails vs. Rails+Angular.

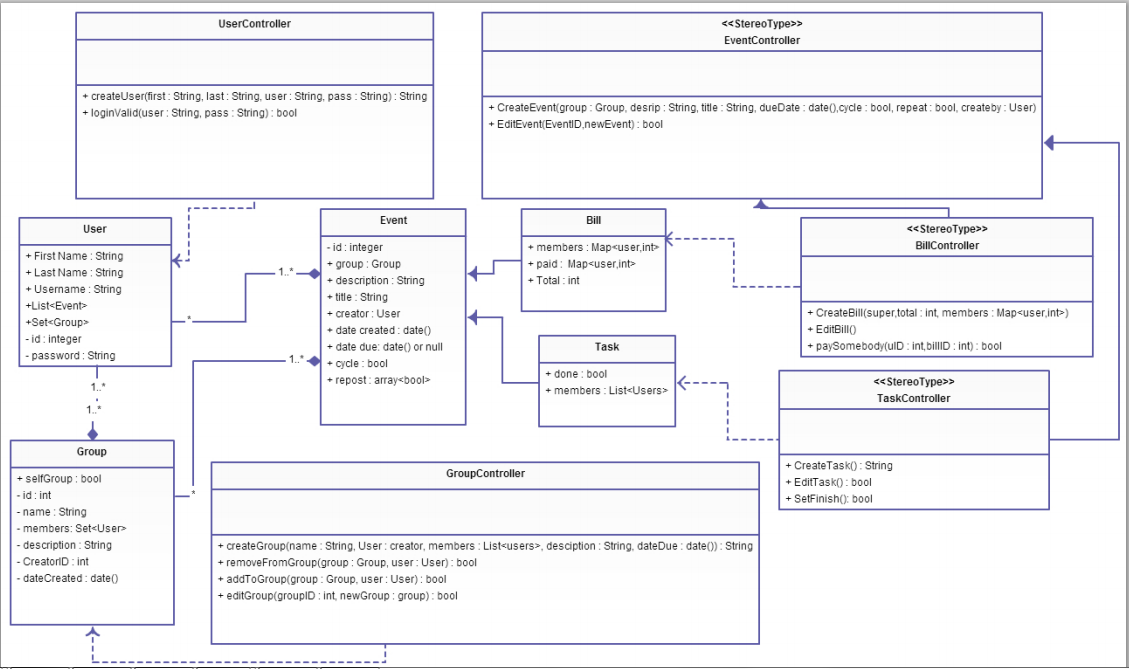
* We will use the MVC pattern with the help of Angular.
* The controllers will know about the views and models, and will be a “thin” controller--just serves to connect the views to the models.
* We will have four main models--the GroupModel, UserModel, TaskModel, and EventModel. Then, we will have one controller for each view (page), and those controllers will call appropriate methods in the models to update them.
* The models will in turn make AJAX requests to the backend controllers to update the databases, or pull information from them. This information will be packaged into the frontend model classes--User, Group, Task, Bill, and sent back to the controllers/views

EDIT: We have changed more of our design. Here are a few changes:

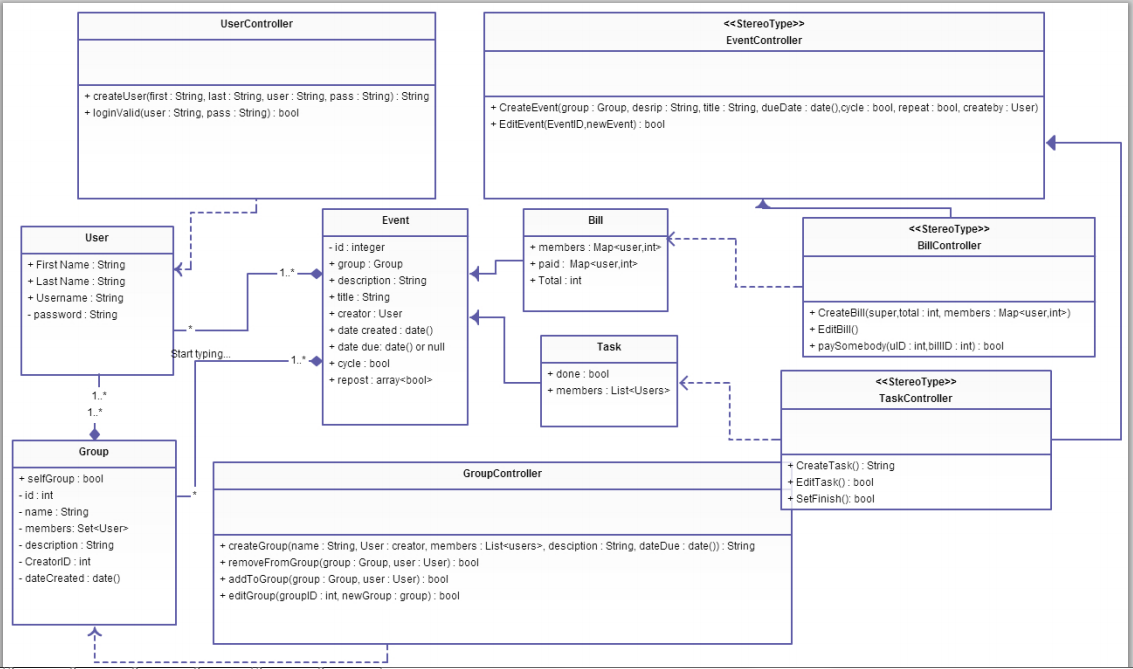
* Self Groups
  + TeamPlayer revolves around groups, and when a user first signs in, they need to be a part of a group before they can do anything. We realized this flaw during our first customer prototyping test, when the user wanted to add a task, but didn’t have a group to choose. So, every new user already belong to a group with just themselves. This way, they can do things before they have created other groups, or added other users.
* Bill Un-cycling
  + When further thinking about bills between roommates, bills are often sent to one respective person, who pays for them. It is rarely cycled to another roommate since that would make it complicated for the electric or water companies. Therefore, bills will no longer cycle between group members. If necessary, new bills can be created individually by each member.
* Overdue Task Constraints
  + Since tasks can cycle and members can become irresponsible, it would be unproductive to extend their deadline. The task would remain marked as incomplete, but would cycle to the next member, because it would be useful to notify the next member that something needs to be done.
* New Task Generation
  + Individual tasks are marked as cycling or repeating on the back end, but we hadn’t planned out how exactly this would appear to the user. We decided that weekly, there would be an update of each user’s tasks, so a task from a cycle would be a new object.
* Un-inheritance of Tasks and Bills
  + Originally, we saw tasks and bills as very similar classes, and therefore made them extend from and Event class. However, with further implementation and better specifications, we realized how different both are, and decided to make them their own individual classes that do not extend from any higher class.

**Class Diagrams**

**Back End Database UML Diagram**

****

**Front End UML Diagram**

****

In our UML class diagram for the backend, we have separated out functions into “Controllers” classes, rather than just having those methods in the Group, Bill, etc. classes. We did this because when you send an Ajax request to Rails, by default, it sends the request to the appropriate controller (in Rails), so it is necessary to define the functions in the diagram for each controller class. Also, the objects Group, Bill, etc. are actually in the databases, just represented as models in rails--so we cannot write functions there. Our UML class diagrams for back end and front end are very similar with the minor difference that users need only provide very little to be able to access their account. It is necessary that the front and back end modules be similar so that they can communicate with JSON objects easily.