SmartMate

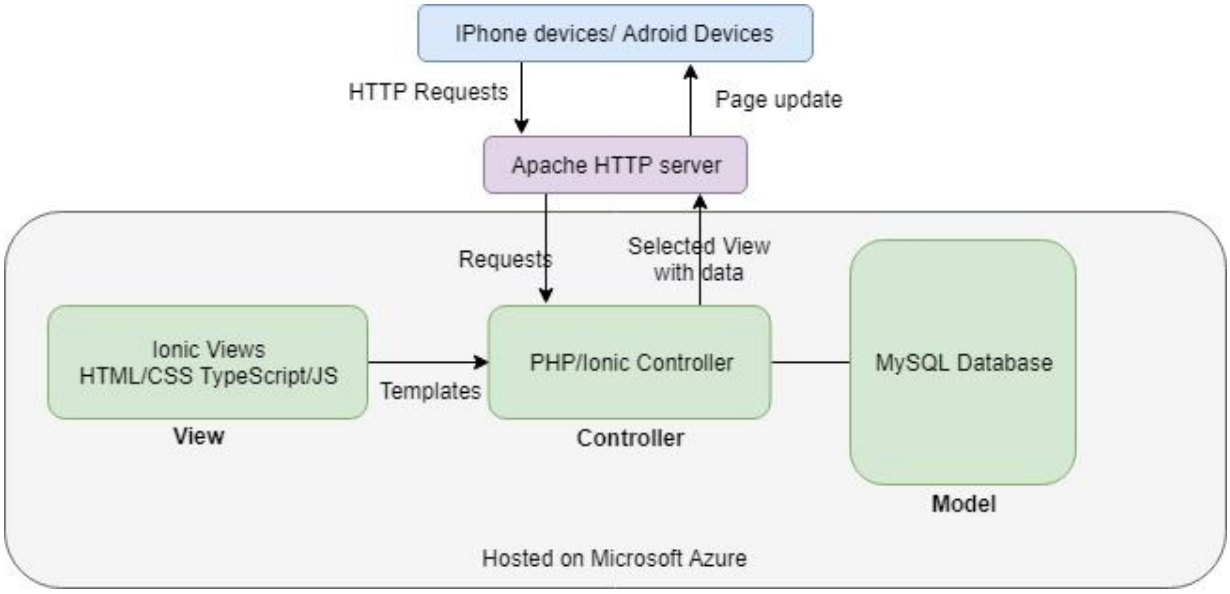
Design and Planning Document
2018-10-12, version

Document Revision History

Rev. 1.0 <2018-10-7>: initial version

Table of Contents

System Architecture



Model-View-Controller (MVC)

a. **Models**

Models contain business logic of the application such as schema for users, groups and chores and relations. It should be separated from the view and respond to controller with requested data. The process will be implemented by PHP applications which communicate with MySQL database based on controller's inputs.

b. **Views**

The views determine what is presented to users on mobile devices and how it is presented graphically. We will construct our views on Ionic framework with HTML/CSS. JavaScript/TypeScript will also be used to communicate either Ionic controllers or PHP controllers. The data interchange format might be in HTML or in JSON format.

c. **Controllers**

Controllers are responsible for processing user's inputs (HTTP requests), invoking functions in Models, embedding returned data with selected view and send views back to Apache server, then to user's mobile devices. We have two ways to construct our controllers:

i. Ionic framework

Ionic framework provides the flexibility to create controller methods in each module, selecting ionic views and processing requests with PHP applications in model.

ii. PHP

We tend to use PHP to construct our controller because we are to make our Model function in PHP. We believe the communications between controller and model could be more intuitive if we have a PHP controller.

Design Details

Front End

1. **Mobile Application Key Components:**
    a. Ionic
    b. CSS3
    c. HTML5
    d. JavaScript
    e. NodeJS
    f. Cordova

    In discussing which device/landscape to create *SmartMate* it quickly became clear that from our user stories and customer input, a mobile application would make the most sense. Our customers' thought process were along the lines of tracking chore completion and bill splitting should be a quick, seamless experience, available where ever, when ever they want. Naturally, this aligns with the strengths of a mobile device, however, the next question became which mobile platforms to develop for.

    In preliminary discussions of which landscape to support, the team went through pros and cons for developing for each of the two main mobile operating systems: iOS and android. It quickly became apparent that developing a native application for both iOS and android was not a feasible task in the short amount of time we have, rather we would have to choose one landscape over the other or look into different solutions. The team looked into both options and decided that, in order to reach the largest customer base and create an application readily portable to future platforms, a cross-platform option should be pursued. After research into options available for such a development, the front-end team settled on Ionic.

    The front end of the application will utilize Ionic, a cross-platform solution for mobile development. Ionic works by wrapping web apps written in HTML5, JavaScript, and CSS3 in wrappers so they can be run natively in the landscape of choice, be iOS or android. Ionic's command line interface (CLI) is based off of Node's node package manager (npm), hence the need for NodeJS in this project.

Furthermore, Ionic is built on top of Apache's Cordova, thus while development of *SmartMate* will utilize Ionic directly, Cordova will be utilized indirectly.

Essentially, Ionic is a library of Javascript and CSS3 that allow for the look, feel, and functionality of a mobile application, as opposed to a website. After the the user interface has been created, the Cordova component of Ionic packages the components, allowing for native execution on the platform of choice. There are not many solutions that offer similar functionality to Ionic, however React native was a different route we could have gone. After further research, we concluded that the learning curve for Ionic would be smaller, so in a project pressed for time, we chose to move forward with Ionic and all of the external interfaces it utilizes.

Back End

1. **Database**
    - **a.** MySQL is a viable option at this point to store all user data; being dynamic and open source, it lends itself well to collaborate with our Frontend use of Ionic. The highly structured environment of this object relational database system has the advantage of straightforward and intuitive syntax. Compared to SQLite, MySQL achieves better performance in terms of concurrency and is less of an "out-of-the-box" database resource. Despite the easier arrangement of SQLite, we plan to use MySQL for its network access, degree of concurrency, and buffered memory storage.
    - **b.** Building upwards our MySQL database, we will use PHP to perform accesses. **PHP: Hypertext Preprocessor** is specially designed for web development and should prove to be a worthy investment of our time in deploying applications. It has the advantages of being an open source interpreter (real-time) and dynamic. The scripting language itself is powerful, free, and plays well with JSON formatted data in sending and receiving requests. Upon gaining access to the PHP web server, we will need to write a PHP web service to query our database.

2. **Hosting**
    - **a. Heroku**
        - i. PHP is compatible with the PaaS Heroku, and this will add a nice advantage with the Cloud capabilities of Azure or AWS. Heroku will accomplish many details, such as all server configurations, caching, data compression and memory allocation for our app. It also supports SQL, which was attractive to our team for our database decision.
        - ii. With this, our app will be built, run, and operated on the cloud. The free version provides 512 MB of RAM, so we may investigate greater options if necessary. The frontend decision to use JavaScript (via Node.js), will play well with this hosting service.

iii. We may even be able to use Heroku Postgre for our database (DBaaS). This is heavily based on SQL; learning it should be reasonable for teammates.
iv. The source code of our application will be sent to Heroku using Git
v. *Dynos* are isolated, virtualized unix containers and serve as the run time environments
vi. Heroku uses a random selection algorithm to distribute HTTP/HTTPS load balancing across these web dynos and is predicted to easily accommodate our app's use

b. **Azure**
   i. We may use the global network of Microsoft-managed data centers as a cloud platform in conjunction with Heroku.
c. AWS
   i. Heroku services are currently hosted on Amazon's cloud platform, and therefore this may be a likely choice to host our application on the cloud in case we decide to abandon Azure.

3. Framework
a. Laravel
   i. Here, we examine a free, open-source method of developing applications. Laravel was designed by Taylor Otwell and is based on an additional PHP framework known as Symphony. It follows a model-view-controller (MVC) design pattern in its architecture. Scalable and preserving of namespaces and interfaces, it will aid in our organization and management of resources. Moreover, Azure app hosting server typically works with Laravel to support PHP + MySQL apps, for which there exist detailed tutorial pages. Therefore, Laravel has the potential to help backend integrate with Servers.

Risk / Limitations associated with design

1. Front-end:
   a. Ionic has a small performance gap comparing to native app.
   b. Some functions may not be available in the Ionic framework. We might have to develop some plugins to create our own functions or adjust our design patterns based on what ionic could provide as components.
   c. The payment in Bill management might be hard as we might not be able to add payment protocol to our app.
2. Back-end:
   a. Connect database to front end. Users interact with database heavily through front end interface. It is crucial to setup a reliable connection between application and database.

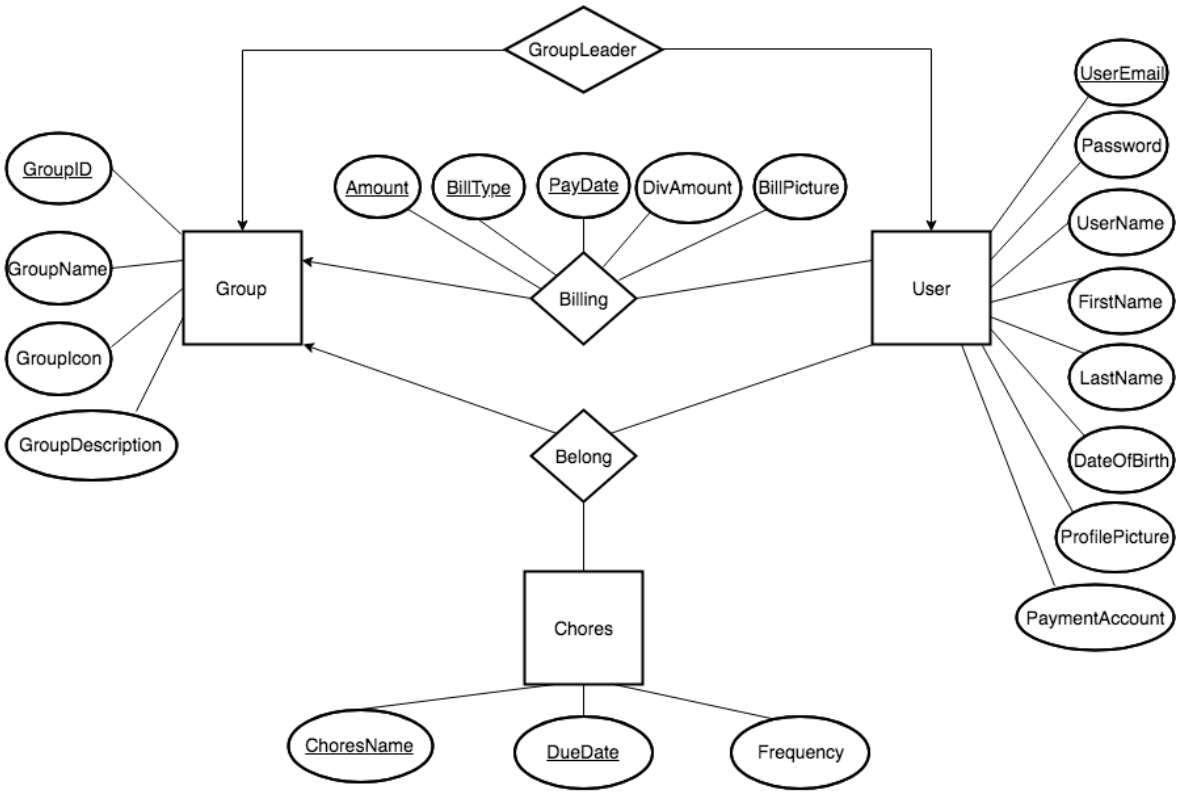
b. Migrate database to cloud. Hosting service providers have different specification for their cloud instance. We need to choose a host that is most reliable and gives us easy way to set up database. After setting up database on cloud, it might involve further complications because cloud environment is different from local environment.

Alternative Design

1. Instant Message System (IMS) might be an alternative design that we can add in our app for later iterations. IMS may help users to communicate with each other easily when they have some chore or payment related issues. When a chore is reassigned or an invitation is sent between users, instead of pushing notifications, those kind of information will go into instant message system. But we still need to do more market research to see whether our users will use our in-app communication system or other message app such as Messengers/Whatsapp or imessage.

2. In the early iterations, we plan to use Venmo/Paypal/Apple Pay link in the Bill Management System, which means that users have to jump to another platform outside of our app to complete the payment. We consider add payment protocol within our app as an alternative design which may better improve user experience for bill management as they can stay in the app to finish all steps from issuing, recording and splitting a bill payment to complete the payment.

DataBase ER Diagram



User(<u>UserEmail</u>, Password, UserName, FirstName, LastName, DateOfBirth, ProfileImage, BillAccount)
Primary Keys: UserEmail

Group(<u>GroupID</u>, GroupName, GroupIcon, GroupDescription)
Primary Keys: GroupID
Foreign Keys: Member (User.UserEmail)

Chores(<u>ChoresName</u>, <u>DueDate</u>, Frequency)
Primary Keys: ChoresName, DueDate;
Foreign Keys: GroupID (Group.GroupID), Assignee (User.UserEmail)

Billing(<u>Amount</u>, <u>BillType</u>, PayDate, DivAmount, BillPicture)
Primary Keys: Amount, BillType, PayDate;
Foreign Keys: GroupID (Group.GroupID), Payer (User.UserEmail)

Protocols

SmartMate will use HTTPS protocols in order for any client to communicate with our Apache HTTP server. GET and POST commands provide a means of requesting pages and dynamic page content. JSON will be the preferred format of all responses, and sensitive user information will be kept secure by these protocols.

Algorithms

1. **Searching**

   Searching will be done by MySQL database queries. Once user's input email is received by the controller, PDO function, query(), will be called to search any tuple with the corresponding email through SQL commands:

   SELECT * FROM Users WHERE UserEmail = *user inputs*;

2. **Input validation**

   Input Validation will be done by Regex. Different types of inputs should be validated based on following rules:

   a. Email/UserID should follow the regular pattern of any existing email formats
      i. **/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/**
   b. Password is in a range of 6-20 characters with special characters
      i. **/^[A-Za-z0-9!-@#$%^&*()_]{6,20}$/**
   c. First/Last Names only allow alphabetical characters in a range of 2-20 characters
      i. **/^[A-Za-z]{2,20}$/**
   d. Group Names allow word characters, '_', '-' in a range of 2-20 characters
      i. **/^[A-Za-z-_]{2,20}$/**

3. **Item Priority Based on Scheduling**

   Each chore item and payment due item will have its DueDate. The DueDate follows MySQL's timestamp format, YYYY-MM-DD HH:MM:SS. The priorities of items will be sorted based on their due date's real time order.
   For example, 2018-09-01 05:00:00 is earlier than 2018-09-01 10:08:30. Therefore, the item with former due date will show up with the higher priority.

Security Framework

Security measurement is needed to counter malicious attacks or abuse of system resources.

reCaptcha:

To prevent bots from signing up and spamming the system, reCaptcha is required in Sign-Up process. *[In]visible ReCaptcha for iOS* is an implementation of Google's reCaptcha on iOS device. Normally, ReCaptcha validates the challenge automatically and retrieves a token using JS API. When the API can't ensure the user is human, a challenge is presented. For example, calculating the result of "2ol +lo" challenge will be presented.

Google Authenticator:

Google Authenticator provides support for 2-setup authentication. When user wants to access sensitive data such as bank account information, he/she might be required to enter PINs from authenticator to ensure his/her identity. Google Authenticator offers denpendies and API for two-factor authentication.

Security concern for database:

SQL Injection:

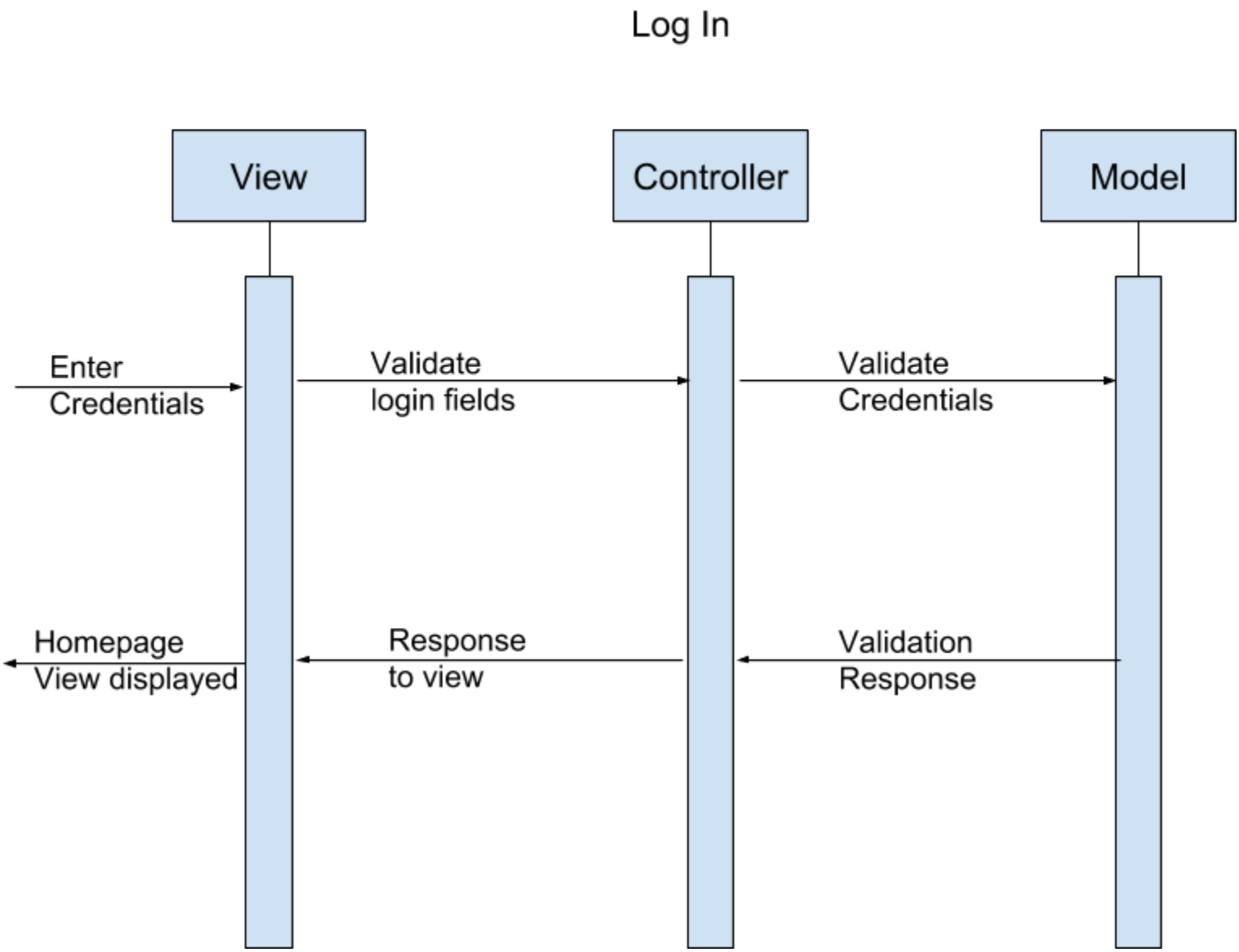
SQL injection is a code injection technique to undermine database. To prevent SQL injection, we should:

1. Use prepared statements and parameterized queries.
2. Use Whitelist input validation.
3. Use stored procedures.

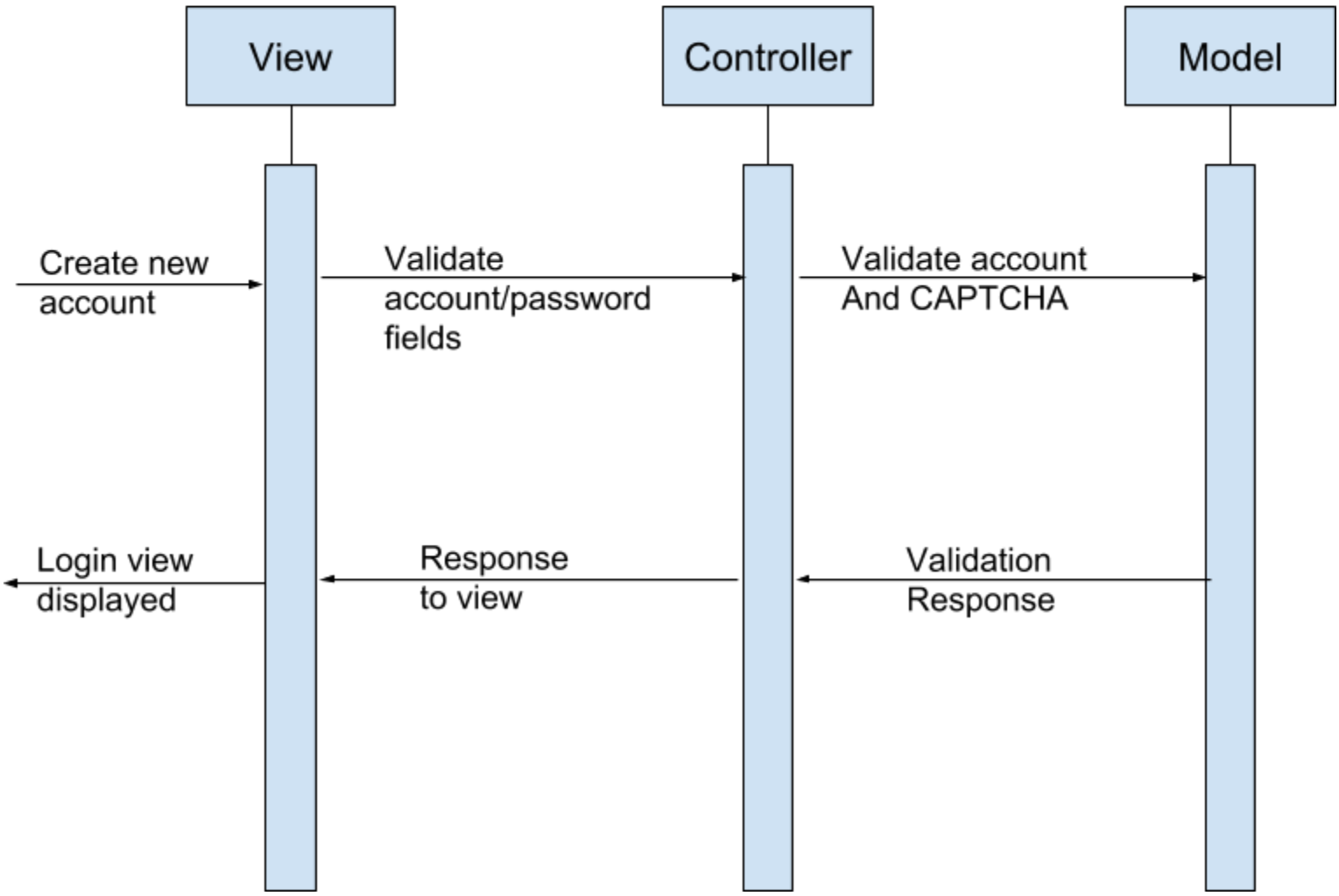
Password Hashing:

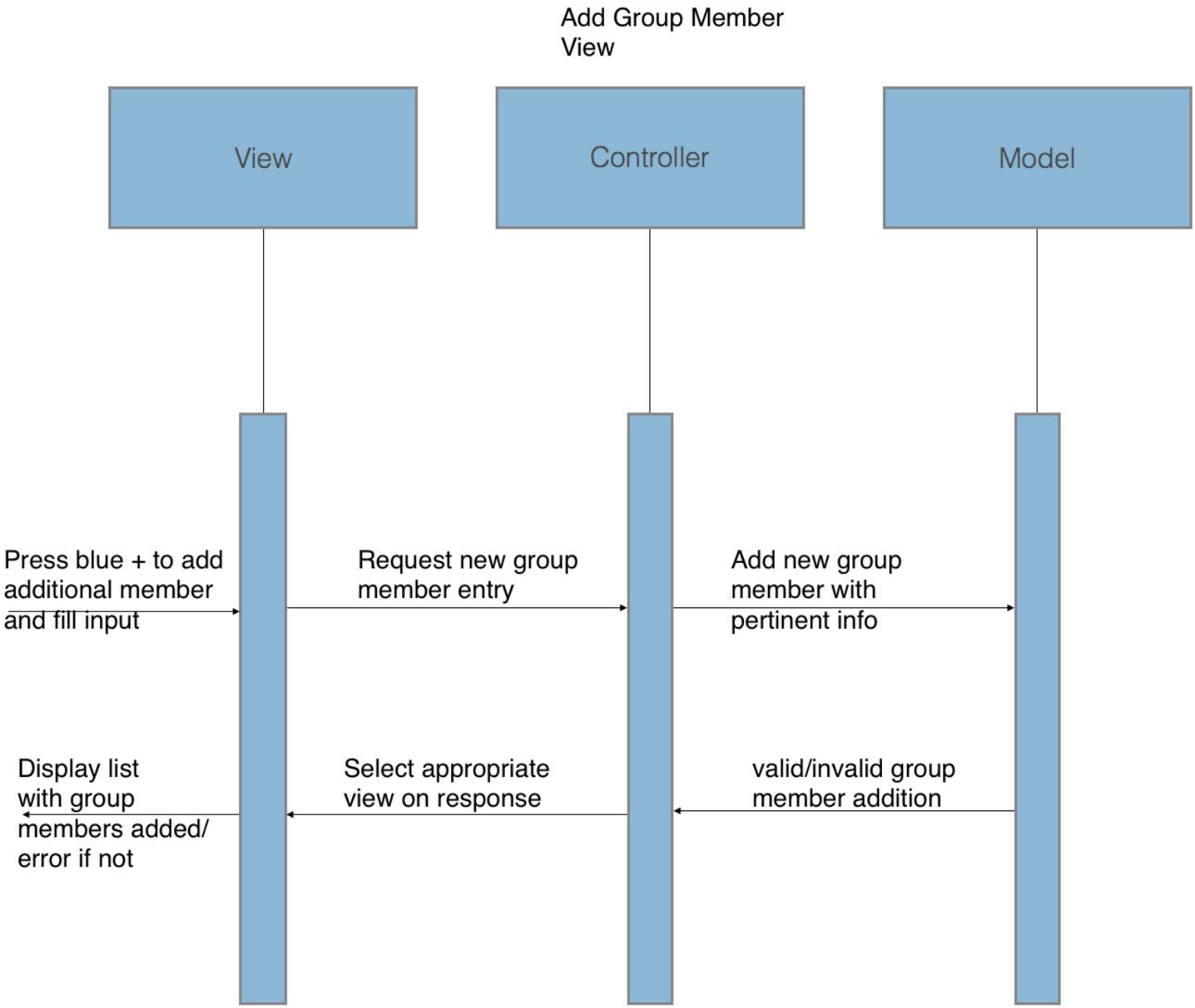
User password should be stored securely. To achieve this, we apply one-way hash before storing user password into database. Even if our database is compromised, attacker can't tell the exact value of user's password since it is hashed.

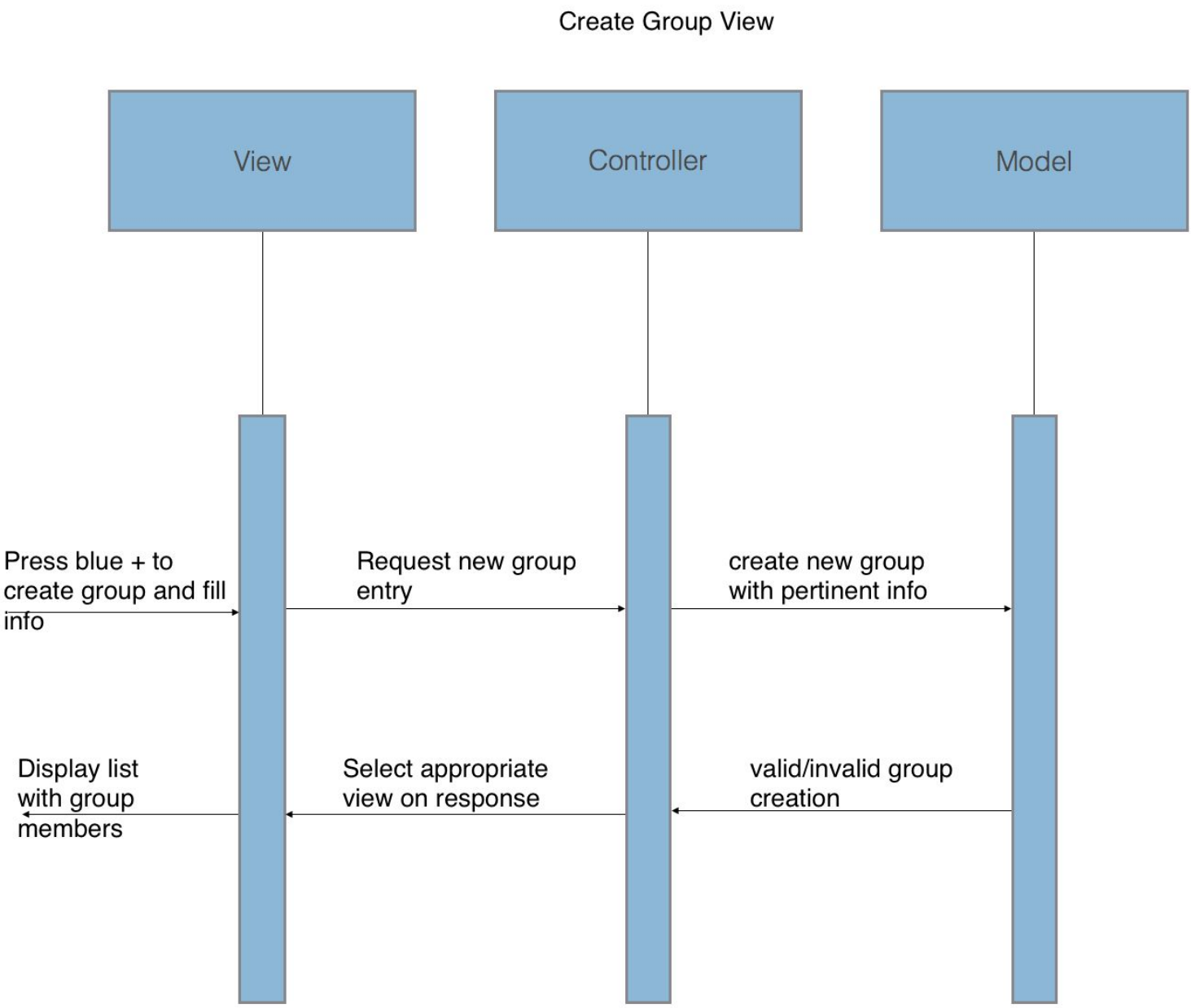
Sequence Diagrams

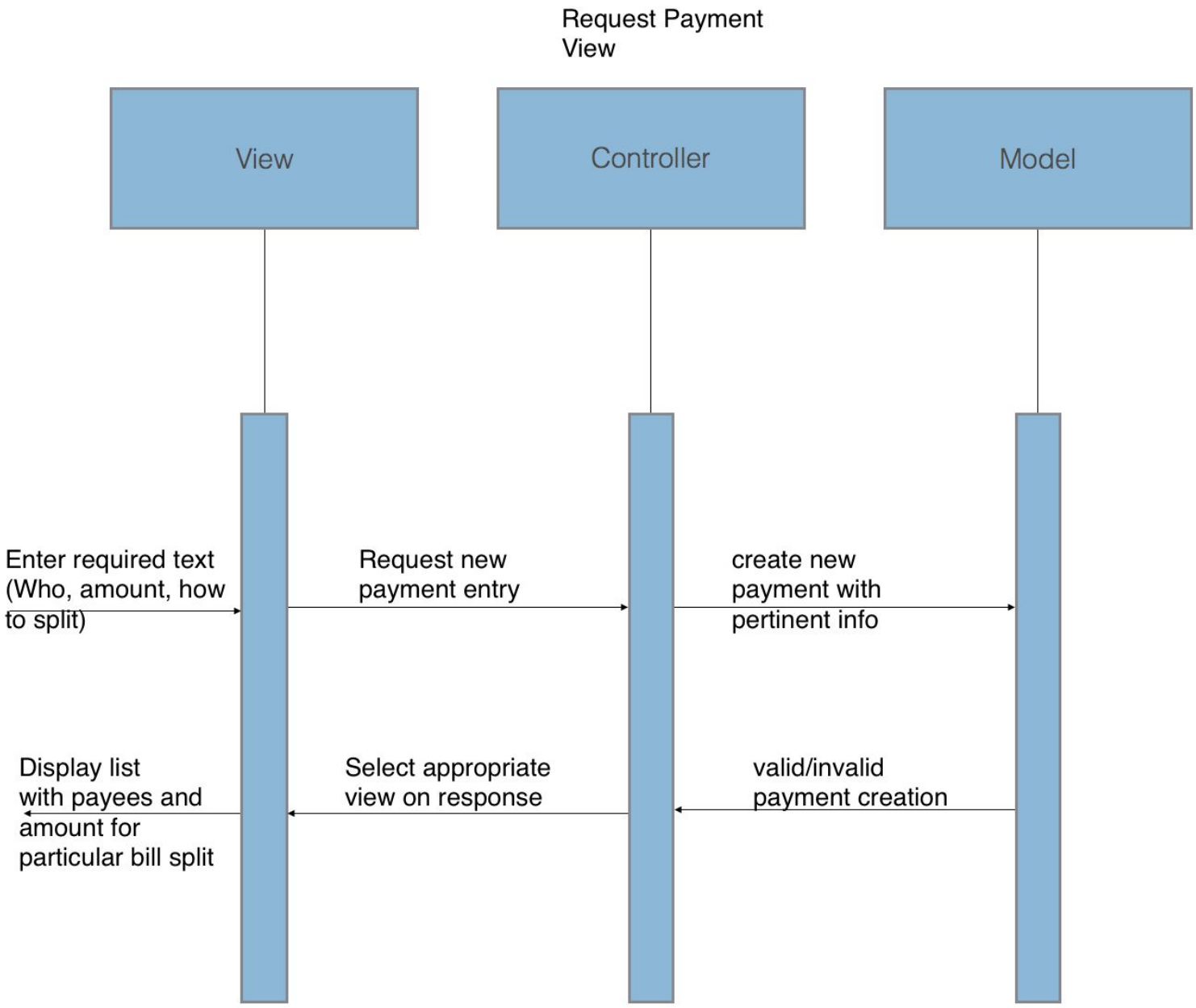


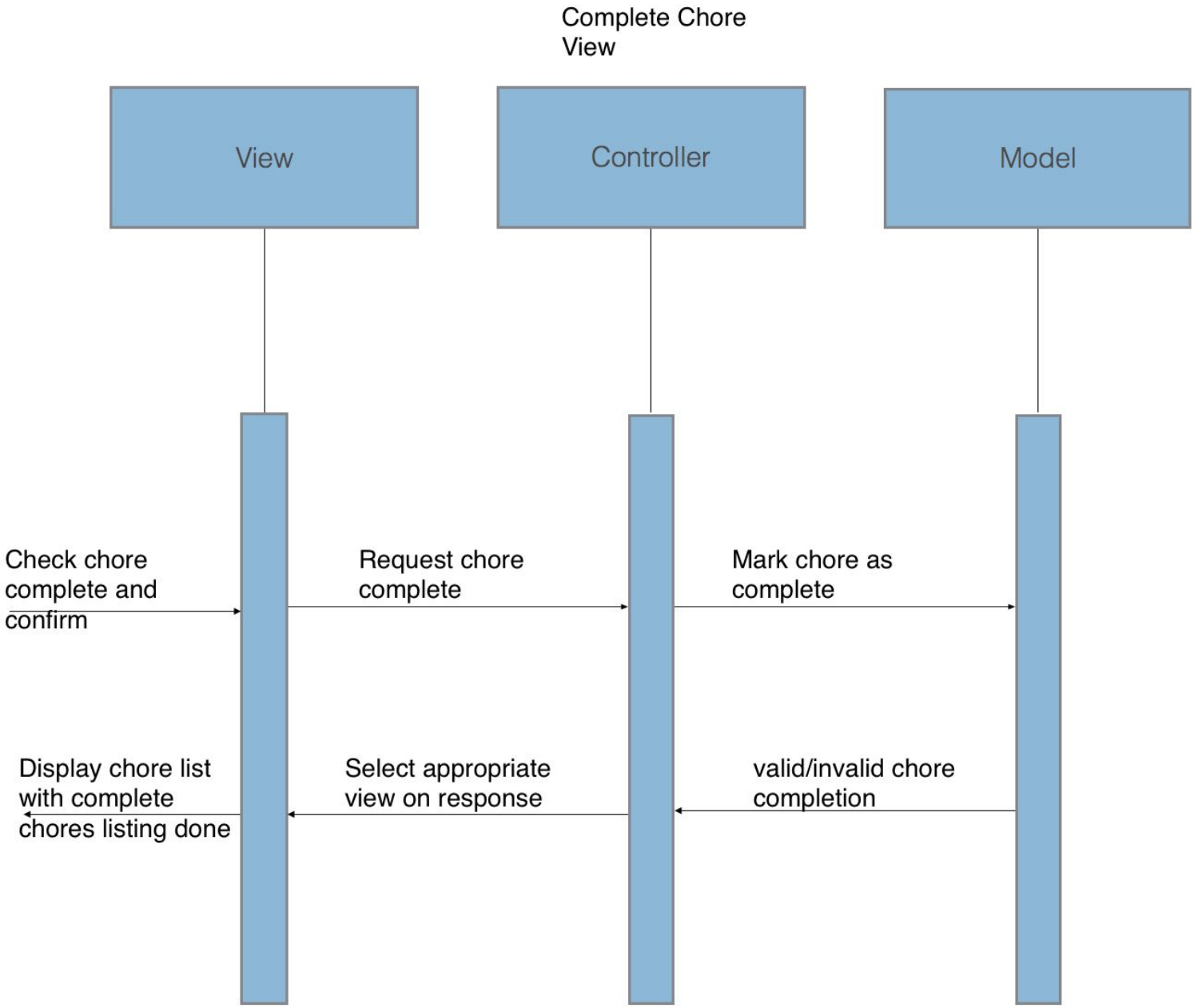
Sign Up


View
Controller
Model
Create new account
Validate account/password fields
Validate account And CAPTCHA
Login view displayed
Response to view
Validation Response

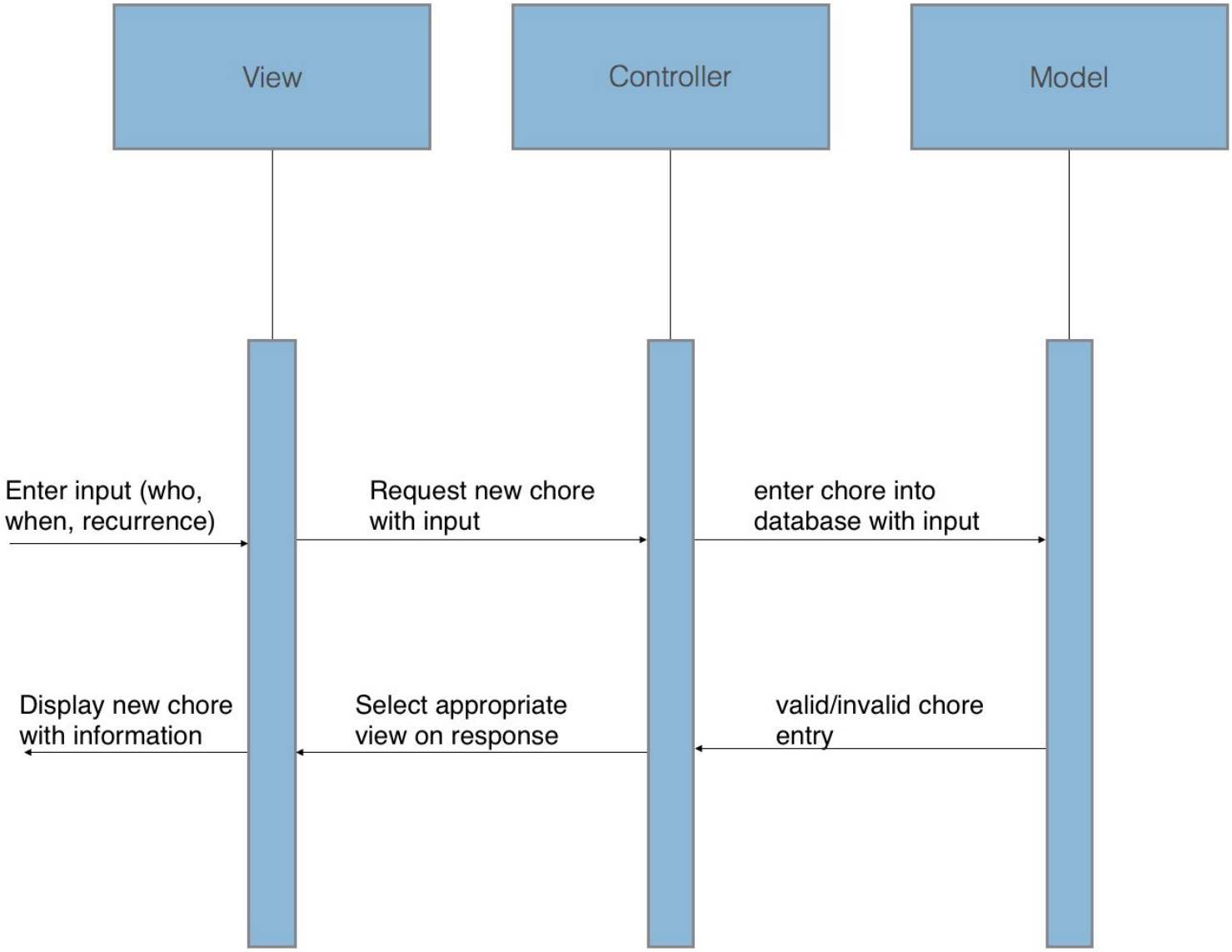

Add Group Member
View
View
Controller
Model
Press blue + to add additional member and fill input
Request new group member entry
Add new group member with pertinent info
Display list with group members added/ error if not
Select appropriate view on response
valid/invalid group member addition


Create Group View
View
Controller
Model
Press blue + to create group and fill info
Request new group entry
create new group with pertinent info
Display list with group members
Select appropriate view on response
valid/invalid group creation

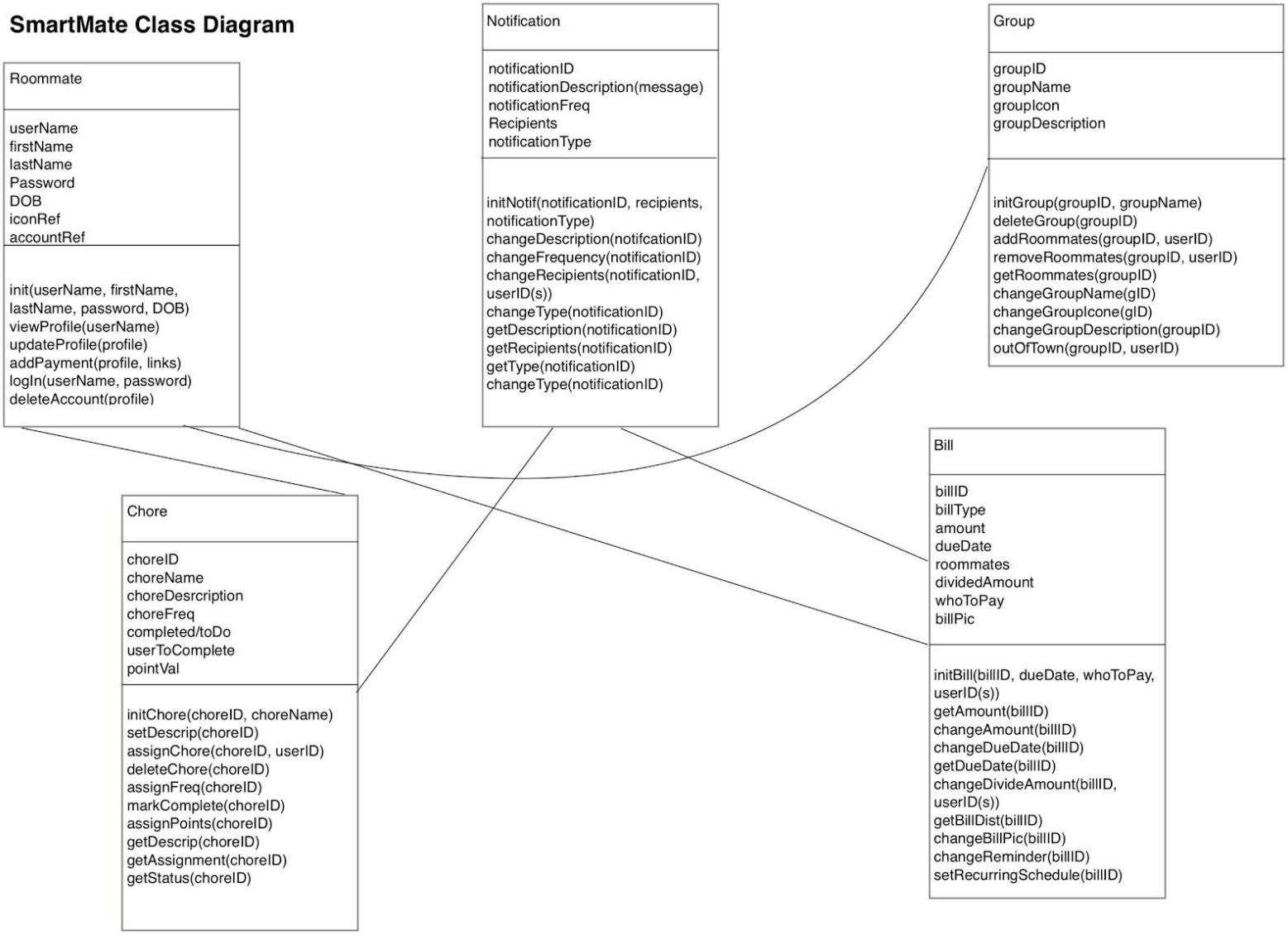

Request Payment
View
View
Controller
Model
Enter required text
(Who, amount, how
to split)
Request new
payment entry
create new
payment with
pertinent info
Display list
with payees and
amount for
particular bill split
Select appropriate
view on response
valid/invalid
payment creation


Complete Chore View
View
Controller
Model
Check chore complete and confirm
Request chore complete
Mark chore as complete
Display chore list with complete chores listing done
Select appropriate view on response
valid/invalid chore completion

Create Chore View



Class Diagrams



Invariants and Class Description:

1) Roommate - The roommate class is the parent class for all roommates, regardless of in-app privileges. The roommate class will include all information pertinent to each roomate in the group including:

1. Attribute Description
    a. userEmail
    b. First Name
    c. Last Name
    d. Password
    e. Date of Birth
    f. Reference to Profile Image
    g. Payment Accounts Information and link (Venmo/Apple Pay/Paypal)
2. Method Description
    a. init(userEmail, First Name, Last Name, Password, DOB, Reference to Image):
    Initializes a roommate user with the listed parameters.
    b. viewProfile(userEmail)
    Given a profile, will show pertinent profile information like name, image, email/username...
    c. updateProfile(userEmail, First Name, Last Name, Password, DOB, Reference to Image, links and account/credit card info)
    Given a profile, allows functionality for users to update aspects of their profile, including the ability to change payment methods on his/her profile such as Venmo/Apple Pay accounts.
    d. addPaymentMethod(userEmail, links and account/credit card info)
    Allows users to add a payment method to his/her account such as Apple Pay/Paypal/Venmo
    e. logIn(userEmail, Password)
    Functionality to log in to a particular account if the input password is correct.
    f. logOut(Username)
    Functionality to log off of a particular profile
    g. deleteAccount(userEmail)
    Functionality to delete a given account

2) Group - The Group class will be comprised of Roommates.. It will contain all of the methods that pertain to group functionality and management.

1. Attribute descriptions
    a. groupID
    b. groupName
    c. groupIcon
    d. groupDescription
2. Method Descriptions

a. initGroup(groupID, groupName)
   Initializes a group with aforementioned parameters
b. deleteGroup(groupID)
   Allows for the removal of an existing group
c. addRoommates(groupID, userID)
   Add a roommate to a specified, existing group
d. removeRoommates(groupID, userID)
   Remove a specified user from an existing, specified group
e. getRoomates(groupID)
   Given a specified group, returns every group member
f. leaveGroup(groupID, userID)
   Given a specified, existing group member, removes him/her from specified, existing group
g. changeGroupName(groupID)
   Changes group name of specified, existing group
h. changeGroupIcon(groupID)
   Changes group icon of specified, existing group
i. changeGroupDescription(groupID)
   Changes group description of specified, existing group
j. reassign(groupID, userID)
   Marks specified member of group as out-of-town, putting in a dormant state so no chore can be assigned

3) Chore - The Chore class aims to address any functions and scenarios that pertain to managing chores within a group.

1. Attribute Description
   a. choreID
   b. choreName
   c. choreDescription
   d. choreFrequency
   e. Completed/ToDo
   f. userToComplete
   g. pointValue
2. Method Descriptions
   a. initChore(choreID, ChoreName)
      Initializes a chore with the aforementioned parameters
   b. setDescription(choreID)
      Given a particular chore, set the description of what the chore entails
   c. assignChore(choreID, UserID)
      Given a particular user and chore, assigns the user to the chore
   d. deleteChore(choreID)
      Allows for a chore to be deleted after being added
   e. assignFrequency(choreID)

Given a particular chore, enables the frequency of how often the chore should repeat itself on a daily/weekly/monthly/yearly basis

f. markComplete(choreID)
Allows a roommate to mark a chore complete and/or submit a photo for evidence
g. assignPointValue(choreID)
Assigns a point value to each particular chore, user is awarded the points after a task is marked complete.
h. getChoreDescription(choreID)
Given a particular chore, returns the description of the chore
i. getChoreAssignments(choreID)
Given a particular chore, returns how often and to who a chore has been assigned to
j. getChoreStatus(choreID)
Returns the status of a chore (Not complete, working on it or complete)

4) Bill Management - the bill class will be composed of different bill objects that include how much a bill is, with whom the bill is being split amongst, when the bill is due, and the ability to take a picture of the bill for documentation.

1. Attribute Description
    a. billID
    b. typeOfBill
    c. Amount
    d. payDate
    e. Roommates
    f. dividedAmount
    g. whoToPay
    h. picutreOfBill
2. Method Descriptions
    a. initBill(billID, amount, payDate, whoToPay, dividedAmount, userID(s))
    Creates a new bill object with aforementioned parameters
    b. getAmount(billID)
    Gets the *total* amount for a given Bill
    c. changeAmount(billID)
    Given a bill, changes the amount of a bill still owed
    d. changePayDate(billID)
    Given a bill, changes when a certain bill is due
    e. getPayDate(billID)
    Given an existing bill, gets the pay date of a particular bill
    f. changedivideAmount(billID, userID(s))
    Given a bill, change the percentage of who owes what for a particular bill
    g. getBillDistribution(billID)
    Returns the amount owed by each roommate the bill was shared with.
    h. changeReceiverOfPayment(billID, userID)

Given an existing bill and userID, changes who the payment for a bill is going to

i. changeBillPicture(billID)
Given a bill, allows you to add, remove or change the picture of the bill you want to share.
j. changeReminder(billID)
Given a bill, modify when notifications are sent out for a bill (i.e. day before, hour before…)
k. setReccuringSchedule(billID)
Given a particular bill, set a recurring schedule for the bill (weekly, monthly, yearly…)

5) Notifications - Notifications will handle all notifications related to user activity, chores, bills as well as any other notifiable event.

1. Attribute Description
    a. notificaitonID
    b. notificationDescription
    c. notificationFrequency
    d. recipients
    e. notificationType
2. Method Description
    a. initNotification(notificationID,recipients, notificationType)
    Initializes a new notification using the aforementioned parameters.
    b. changeNotificationDescription(notificationID)
    Given a notification, gives the ability to change the description
    c. changeNotificationFrequency
    Given a notification, gives the ability to change how often the notification is pushed.
    d. changeRecipients(notificationID, userID(s))
    Given a notification, changes the recipients of the particular notification
    e. changeNotificationType(notificationID, notificationType)
    Given a notification, gives the ability to change the type of notification (bill, chore, etc…).
    f. getNotificationDescription(notificationID)
    Given a notification, returns the description (message) of the notification
    g. getNotificationRecipients(notificationID)
    Given a notification, returns who will receive the notification
    h. getNotificationType(notificationID)
    Given a notification, returns the type of notification (bill, chore, calendar event…)
    i. getNotificationFrequency(notificationID)
    Given a notification, returns how often the current notification is set to notify recipients

View Details

List of views:

- **Unlogged View [1]**
- **Log In View [2]**
- **Sign Up View [3]**
- **Home Page (No Group) View [4]**
- **Group Profile View [5]**
- **Add Member View [6]**
- **Home Page View [7]**
- **Member Management View [8]**
- **Create Chore View [9]**
- **Bill Management View [10]**
- **Error Message Views [11]**
- **Confirmation Windows Views [12]**
- **Personal Profile View [13]**
- **Notification View [14]**

1. Unlogged View

Unlogged View appears when the user first downloads the app or logs out the account.

Content	Component	Description
SmartMate Logo	Icon	The logo for SmartMate.
Sign up	Button	Direct the user to **Sign Up View [3]**
Sign in	Button	Direct the user to **Log In View [2]**
About us	Link	Direct the user to the **About us View [12]**

2. Log in View

Log in View appears when the user does not open the app for a long time or when the user wants to switch to other accounts.

Content	Component	Description
SmartMate Logo	Icon	The logo for SmartMate.
Email Address	Input Text Box	Text Box for entering email address

Password	Input Text Box	Text Box for entering password
Sign in	Link	Direct the user to the **HomePage View (No Group/Group) [4] [6];** if the email address or the password is not correct, a error message **(Error Message Views [11])** shows up.

3. Sign up View

Sign up View appears when the user click the sign up button in **Unlogged View [1].**

Content	**Component**	**Description**
Name	Input Text Box	Text Box for entering the name
Date of Birthday (DOB)	Input Text Box	Text Box for entering date of birth
Email Address	Input Text Box	Text Box for entering email address
Password	Input Text Box	Text Box for entering password
Confirm Password	Input Text Box	Text Box for re-entering password
Upload Profile Image	Link	Link to a pop up window that you can upload your image from cloud or local
Payment Account	Link	Link to Venmo/Apple Pay/Paypal
Submit	Button	Direct the user to the **HomePage View (No Group/In Group) [4] [6]**

4. Home Page (No Group) View

Home page view appears when the user has not been assigned to a group.

Content	Component	Description
SmartMate Logo	Icon-Left Toolbar	The logo for SmartMate on the top
Profile Image	Icon-Left Toolbar	Click the image to go to the **Personal Profile View [13]** to edit the user's profile information.
User Name	Text-Left Toolbar	The user's name
Log out	Button-Left Toolbar	Click the button to log out the account
Create a new Group	Text + "+' Button	Direct the user to **Group Profile View [5]**

5. Group Profile View

Group Profile View displays when the user first creates the group or click on the group name on the left toolbar edit the group profile.

Content	Component	Description
Left ToolBar	Several components	See appendix: **Left Toolbar**
Group Name	Input Text Box	Text box for entering group name
Group Description	Input Text Box	Text box for entering group description
Manage Group Member	Link	Direct the user to **Member Management View [8]**
Save	Button	Save all the profile information and send to the back end.

6. Add Member View

Add member view appears when the user click add member **in Group Profile View.**

Content	Component	Description

Left ToolBar	Several components	See appendix: **Left Toolbar**
Search to add new members	Search bar	A search bar to search for users to be added to the group
Search Result	Text	Display search result
Add member	Button on the right of the search result text "+"	Click the button and the new member will receive an invitation (**Confirmation Windows Views [12]).**
Back	Button	Direct to group profile view

7. Home Page View [7]

Home Page View appears once the user has been assigned to a group and whenever the user clicks the SmartMate Icon on the left toolbar, the user is directed to this view.

Content	**Component**	**Description**
Left ToolBar	Several components	See appendix: **Left Toolbar**
Chore List	Text + "+"Button.	The title of the chore. Click the "+" button and direct to **Create Chore View [9].**
Chore List Item	Text Toggle	Display chore list item as: "Date + Task"
Chore List Item Choices	Slide the Chore List Item	1. Slide from left to right: Finish the task; 2. Slide from right to middle: pop out the window which includes the list of members that you can re-assign the task to. Once the user re-assigns the task, the new assignee will receive a notification (**Confirmation Windows Views [12]).** 3. Slide from right to left:

		Cancel the task
Event Reminder	Text + "x" button	Display text to remind you about upcoming special events or notifications. Click the button to clear it.
Create a new Bill	Button	Direct to Bill Management View

8. Member Management View [8]

Content	**Component**	**Description**
Left ToolBar	Several components	See appendix: **Left Toolbar**
A list of members	Text	Display all group members
Add Member	Button	Click the button and direct to **Add Member View [6]**
Dismiss Group	Button	Dismiss the group. Have a pop-out window to confirm the dismiss of the group.

9. Create Chore View [9]

Create Chore View appears when the user clicks the "+" button next to the Chore list title on the **Home Page View.**

Content	**Component**	**Description**
Left ToolBar	Several components	See appendix: **Left Toolbar**
Create Chore	Text	Display Title.
Chore Name	Input Text Box	Text Box for entering chore name.
Assignee + Day in a week (Frequency)	Two blank field with Left drop down menu + Right drop down menu with checkbox	Left drop down menu for choosing the assignee; Right drop down menu for choosing the assign day by clicking the checkbox.

Add New Assignee	Button	Click the button and display a new row of blank "Assignee + Day in a week"
Due Date	One blank field with drop down calendar	Choose due date for the blank field.
Complete the creation of new chore	Button	Save the new chore and update it with everyone's chore list **(Home Page View [7])** in the group.

10. Bill Management View [10]

Content	**Component**	**Description**
Left ToolBar	Several components	See appendix: **Left Toolbar**
Bill Name	Input Text Box	Text box for entering bill name
Bill amount	Input Text Box	Text Box for entering bill amount
Bill Deadline	Blank field with drop down calendar	Choose the date of the completion of the bill
Evenly divide the bill	Check Box	Click the check box and then the bill will be even divided between selected members
List of member	Text + CheckBox + Input Text Box	Click the checkbox of the member if s/he needs to pay the bill. Display the Input Text Box for entering specific amount for each member if the "Evenly divide" Checkbox is not clicked.
Complete	Button	Click the button to send notification (**Confirmation Windows Views [12])** to selected members.

11. Error Message Views [11]

Content	Component	Description
Error Message for email or password	Text + "x" button	Text to tell the user whether they enter the wrong email address or password; click "x" to close the error message.

12. Confirmation Windows Views [12]

Content	Component	Description
Related Confirmation Content for each view	Text	For each pop-up confirmation window, the text will be different. For example, for the bill management view, the text will include bill amount, bill deadline and asking if the user wants to jump to Venmo/apple pay/paypal.
Yes	Button	Click yes to confirm the action
No	Button	Click no to cancel the action
Close	"x" Button	Click "x" to close the confirmation window without confirming any action.

13. Personal Profile View [13]

Personal Profile View appears when the user clicks the profile image **on Home Page.**

Content	Component	Description
Name	Input Text Box	Text Box for entering the name
Date of Birthday (DOB)	Input Text Box	Text Box for entering date of birth
Email Address	Input Text Box	Text Box for entering email address

Password	Input Text Box	Text Box for entering password
Update Profile Image	Link	Link to a pop up window that you can upload your image from cloud or local
Payment Account	Link	Link to Venmo/Apple Pay/Paypal
Update	Button	Update the profile and direct the user to the **HomePage View (No Group/In Group) [4] [6]**

14. Notification View [14]

Content	**Component**	**Description**
Related Confirmation Content for each view	Text	Pop-up window serves the purpose of reminder to user. For each pop-up window, the text will be different. For example, for the billing reminder, the text will include bill amount, bill deadline. For reminder regarding chores, the text will include chores name, due date.
Okay	Button	Click okay to confirm the action

Appendix:

Left Toolbar

Content	**Component**	**Description**
SmartMate Logo	Icon-Left Toolbar	The logo for SmartMate on the top and direct to home page
Profile Image	Icon-Left Toolbar	Click the image to go to the

		Personal Profile View [13] to edit the user's profile information.
User Name	Text-Left Toolbar	The user's name
Group Name	Link-Left Toolbar	Directs to **Group Profile View [5]**
Group members	A list of Links	Display member name for the first iteration. Add instant message function in the second round iteration that if you click on the name, the app opens a message window.
Log out	Button-Left Toolbar	Click the button to log out the account

Implementation Plan

Group structures

Front end - Qiuxuan (W), Anansha (U), Jake (C)
Full stack - Patrick (B), Jiatao (Y)
Back end - Jack (G), Shaoheng (Z), Gong (X)

Iterations

Legend: <Iteration> . <Task#>

Iteration 0 (I0): Set up repository (Due 10/14/2018)
Overview: Repository creation for version control

Task	Priority (1-5)	Difficulty (1-5)	Time Units (1-5)	Dependency	People on Task
1. Set up Git repository	1	1	1	none	Z
2. Fork repositories	1	1	1	I0.1	All

Iteration 1 (I1): Minimum Functional Product (Due 10/28)
Overview: backend setup for HTTP server and host; Model definition, basic user account activity controller, Sign up/Log in/Profile/Home page

Task	Priority (1-5)	Difficulty (1-5)	Time Units (1-5)	Dependency	People on Task
1. Backend set up: configure Apache server; configure Azure server;	1	1	1	none	Z, X
2. Build Models Set up main schemas and relations based on ER diagram (User, Group and Chores) a. Create schemas	1	2	2	I1.1	Z, Y, X
3. Build Controllers for creating accounts	2	2	3	I1.2	Z, X, G

a. Receive user information b. Examine the availability to create corresponding account c. Handle fault input/duplicate account d. Create account					
4. Build controllers for user login a. Receive user login credentials b. Authenticate users	2	2	2	I1.3	X
5. Build controllers for user view profile	3	2	1	I1.8	G
6. Basic user pages for sign up and log in **a. Unlogged View [1]** **b. Log In View [2]** **c. Sign Up View [3]**	1	2	3	none	C, U, W, Y
7. Build **Home Page (No Group) View [4]**	2	2	2	I1.4, I1.6	C, U, W, Y
8. Build **Profile View [14]** a. Build templates for this view b. Set up template relations with DB	3	2	2	I1.7	C, U, W, Y
9. Testing for I1 a. Functionality b. User feedback c. Final User Interface design	4	3	4	I1.1-8	G

Iteration 2 (I2): Main Functions (Due 12/2)

Overview: Group functions, Chores functions development

Task	Priority (1-5)	Difficulty (1-5)	Time Units (1-5)	Dependency	People on Task
1. Build controllers for user group function	3	3	4	I1.7	Z, X, G, Y
2. Implement **Group Profile View [5]**	4	4	4	I2.1	C, U, W, Y
3. Build controllers for user group management	4	3	3	I2.2	G
4. Build **Add Member View [6]** **Member Management View [8]**	5	5	5	I2.1 I2.2	C, U, W, Y
5. Build controllers for user chores function	4	4	4	I2.1 I2.2 I2.3 I2.4	Z, X, G
6. Implement main function **Create Chore View [9]**	5	4	5	I2.5	C, W, Y
7. Build **Home Page View [7]** Complete Home Page	3	2	2	I1.7 I2.1-6	C, U, W, Y
8. Build controllers for user error handling	2	3	3	none	G, X
9. Error prevention development **Error Message Views [11]** **Confirmation Windows Views [12]**	2	3	3	I2.7	C, U, Y
10. Testing for I2 a. Group function testing b. Member	4	3	4	I2.1-9	All

management testing c. Chores function testing d. I2 Document for group, member, chores					

Iteration 3 (I3): Final Product (Due 12/9)

Overview: Implement Bill function, present final product

Task	Priority (1-5)	Difficulty (1-5)	Time Units (1-5)	Dependency	People on Task
1. Build controllers for user billing function a. Type of bill b. Amount c. Paydate d. Divided amount e. Payers f. Picture of bill	4	5	5	none	Z, X, G, Y
2. Implement **Bill Management View [10]**	4	5	5	I3.1	C, W, Y
3. Build controllers for user notification a. Chores reminder b. Bill reminder	2	2	2	I2.8	G
4. Build **Notification View [14]**	2	2	2	I2.9 I3.3	C
5. Build controllers for user personal profile view -User profile modification a. user name b. date of birth c. user email d. save new user information	3	3	3	I1.3 I1.8	Z, Y
6. Build	3	3	3	I3.5	C, U, W

Personal Profile View [13]					
7. Testing a. Create testing task b. Perform full functions testing c. Collect user feedback	4	2	4	I1 I2 I3	C, Y
8. Final review a. Group meeting b. Write review documents	2	1	1	I3.9	All
9. Presentation of final product	4	2	3	I1 I2 I3 I4	All

Testing Plan

Unit Testing - Scheduled concurrently with development, at the completion of important coding milestones and before the final rollout of each iteration.

Unit testing represents testing at the most microscopic level, in other words independently testing different “units” of our code, be individual classes, important algorithms, methods, or code at an even smaller level. While certain test cases can be devised manually, it will be important to automate the process to produce a large amount of test cases in order to catch as many bugs as possible.

Unit testing is applicable in all parts of our development processes— the front end, the back end, and the interface of the two. Catching bugs early on will ensure that future bugs are not compounded and resolving bugs these bugs will remain much more feasible. That being said the following is our plan for testing the frontend, backend, and interface.

Frontend - The frontend is unique from the back end in that a lot of unit testing will be completely visually, making sure the user interface of *SmartMate* “looks right” and “acts right”, meaning shapes and buttons are aligned, read the correct information, and are in working order. Tools that will be utilized for the visual testing will include the iOS simulator in Xcode and the android Emulator.

In terms of debugging, part of the benefit of utilizing Ionic for frontend development is that the application can be easily ported to iOS and Android, and therefore ported into Xcode and google’s suite of development software. From there, we will have access to all of Xcode’s debugging features, as well as Chrome’s DevTools.

Finally, our browser’s built in developer tools have a useful Javascript debugging feature. By including the debugger keyword into the top of certain functions, the browser will halt Javascript execution, allowing us to incrementally step through code within the browser.

Backend - We will begin testing the backend by arranging our database (MySQL) and inserting practice data. After passing all tests that include proper insertion, transformation, and extraction, we will move on to testing other capabilities of the cloud provided by Heroku and the framework Laravel. Our project directory on Laravel will have two test directories: Feature and Unit. Given our development model, these tests will be written using PHPUnit on Laravel before writing the code to pass. Additionally, team members will leverage Laravel’s database testing services.

a. Model Testing
    i. These are unit tests and include the following actions completed on our database models:
        1. Model Creation
        2. Model Deletion
        3. Update
        4. Expectancy of output
    ii. Example using Laravel

```
1. $this->assertDatabaseHas('users', [
            'email' => 'sally@example.com'
        ]);
```

b. Controller Testing
    i. Assert that a specific action on the database model generated a certain sequence of events, e.g. a practice user will be directed to a ‘view profile’ page after creating an app-user entry. The assertion will return either true (success) or false (failure)
    ii. Specific to each ability the controller exercises

Integration Testing- Scheduled concurrently with development, at the completion of important coding milestones and before the final rollout of each iteration.

Integration testing will involve *integrating* our individual units that have been tested and making sure that the interface of the two or more units are behaving as expected. Integration testing will be relative to what an individual unit from unit testing was, be a method or a class, nonetheless integration testing is an important step before system testing, in order to pinpoint and resolve bugs before full system testing. For example, an integration test may be how two views interact, how views are integrating with our database, how classes interact with each other, or any other combination of two or more *units* that have already passed unit testing.

Our integration testing will be completed incrementally, meaning first all front-end and back-end units will be tested separately, where possible. Following the confirmation of expected behaviors, frontend units and backend units will be tested together, starting with fewer units, working up to more and more units involved in each test.

For frontend specific unit testing, there are many tools available to aid with the process, however after some research, Jest will likely be utilized for Javascript testing. Jest includes useful tools like spies, assertions, and mocks that should make testing more convenient than other testing methods. In addition, as in the case of frontend unit testing, there will be a fair of amount of visual inspection for integration testing. Hence, the utility of Jest.

Jest implements a tool called snapshot testing, which essentially takes a screenshot of the current user interface and compares to a reference image— the test will pass if the two images are alike and it will fail if the two images are dissimilar. With this type of utility, we will be able to create scripts with numerous scenarios to complete effective frontend integration testing.

System Testing - scheduled concurrently, and at the completion of each iteration, before rollout to users

System testing will be conducted intermittently throughout development, and at the completion of each major iteration. As previously stated, one of Ionic's strengths is its portability into different platforms. This will enable us to conduct system testing natively on both iOS and android platforms. While emulators exist for both platforms, Xcode's emulator for iOS and android's emulator, we believe the most effective system testing is conducted in a way that mirror's how user's will be interacting with *SmartMate.* Thus, we will conduct system testing on iPhones and android phones that we have available to use.

System testing will involve creating test scripts and verifying expected behaviors in a number of circumstances. At this point, we will also go "hands-on" with the app, using all features that we have created, verifying that all aspects of *SmartMate* are up and running.

Regression Testing- scheduled concurrently with development, especially when new code is written

Regression testing has an overarching goal of ensuring that new additions to any part of the system do not essentially "break" the existing architecture. Regression testing can almost be thought of as modified integration testing, where we are testing existing units with new units. That being said, to perform regression testing, we will re-run past integration tests to ensure expected behavior, as well as write new tests to cover the addition to the architecture.

In particular, we will develop a comprehensive suite that prevents "bugs" from being reintroduced (regression testing) and will only move beyond a certain user story once all its acceptance tests have been passed. This is acceptance testing and is based on acceptance criteria outline in each user story (case).

a. Regression Testing
    i. reuse tests from the previous iteration at the beginning of the succeeding iteration
    ii. Ensures we have a correctly updated project, since bugs become increasingly difficult to address as we move forward
    iii. Any new bugs will be results of current iteration

iv. *Frontend view* testing will be accomplished through the features of Ionic

b. Acceptance Testing
   i. Listed as part of a user case
   ii. We hope to use the automated test suite of Laravel to support all of our backend systems work

Performance Testing - Scheduled concurrently at the completion of iteration 2, 3.

We will be running application by a single device and multiple devices to test on lagging issues, freezing issues and any potential fatal bugs caused by activities that involves multiple users. We will also test on cases to find out limits for chores and group member capability. This will prevent the cases where certain users require huge amount of chore/group members, causing lagging issue or potential bugs. Further, we also will come up with test cases that initiate exhaustive search on database, which might result in lagging respond.

Compatibility Testing - Scheduled concurrently with development

As stated, testing will be performed on simulators and emulators of both of our supported platforms, android and iOS. In addition, we will complete testing on physical devices. Finally, testing will be completed on versions of iOS 7 and up, as well as android 4.4, as stated in our requirements and specifications.

Usability Testing - Time permitting, scheduled after the completion of iteration 3

Time permitting, we will perform usability testing with our customers. Methods that we may employ include cultural probe, contextual inquiry, and heuristic evaluation. The goal of usability testing is to determine how our end users are interacting with *SmartMate*, what is working, what is confusing, and what is holding users up. The results of usability testing can be used to improve upon user interface elements in any future iteration of *SmartMate.*