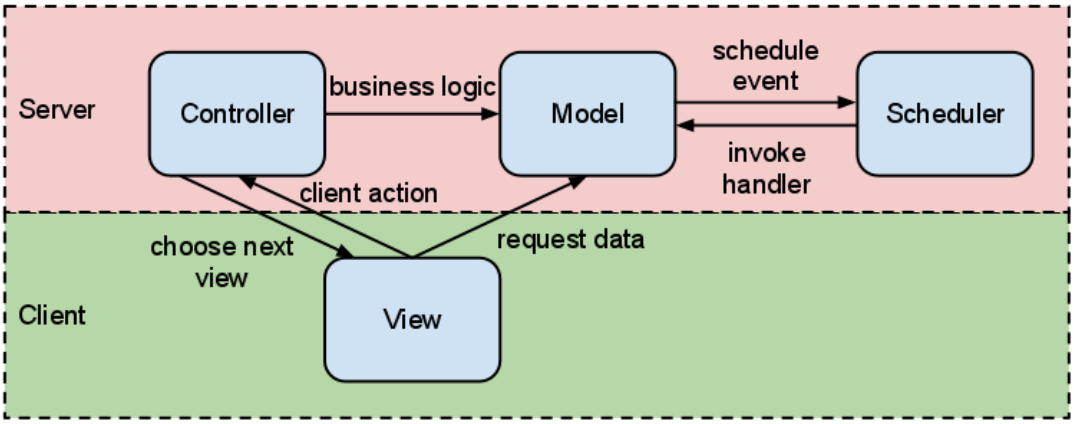
Workshift Manager's Assistant (WSMA)
Design and Planning Document
11/23/2011, version 1.2

Document Revision History

Revision 1.2 2011-11-23

System Architecture



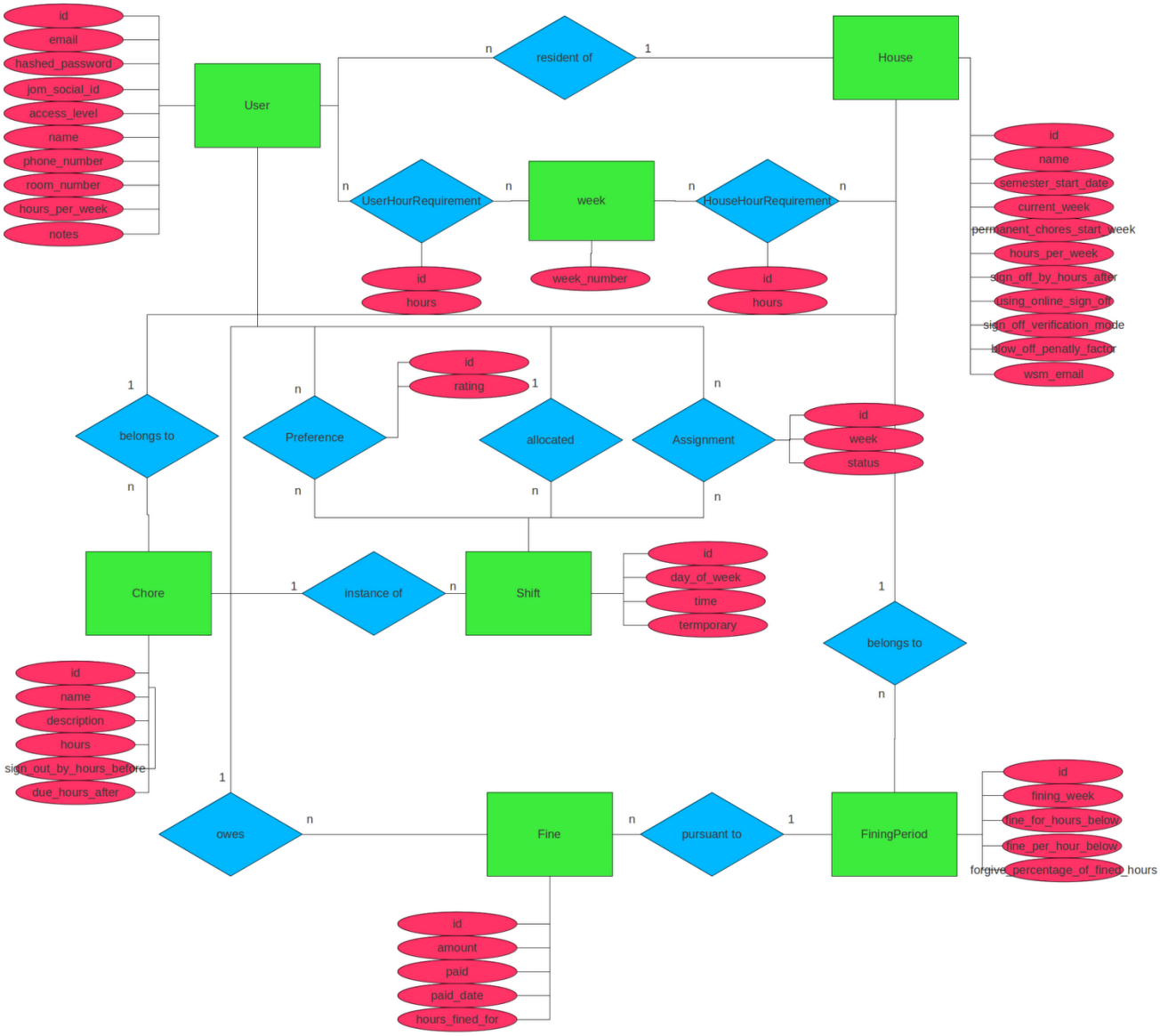
The WSMA is a web application built on the client-server model and the MVC architecture using the Rails 3 framework. The model and controller are implemented on the server side, while the client interacts with the views, which are created by the server and then sent to the client. The views communicate with the controller over HTTP. Our Rails application will be deployed using Passenger onto an Apache web server.

- The model will store data in a Sqlite3 database, and will use the ActiveRecord OR mapper as a database abstraction layer. Each table will be represented by a class in the model, and ActiveRecord will automatically take care of constructors, destructors, accessors, primary keys, and database relationships. This allows the rest of the model to focus on the business logic, ignoring the details of the database. (These implicit details are not listed below in the design details section.) Rails supports this configuration very naturally, and can generate scaffolding to configure the model largely automatically.
- The views will be rendered by a browser by the client, and will consist of html views animated by AJAX and javascript, including JQuery.
- The controller manages incoming requests from the client, consults the model for data and business logic, and chooses the next view.

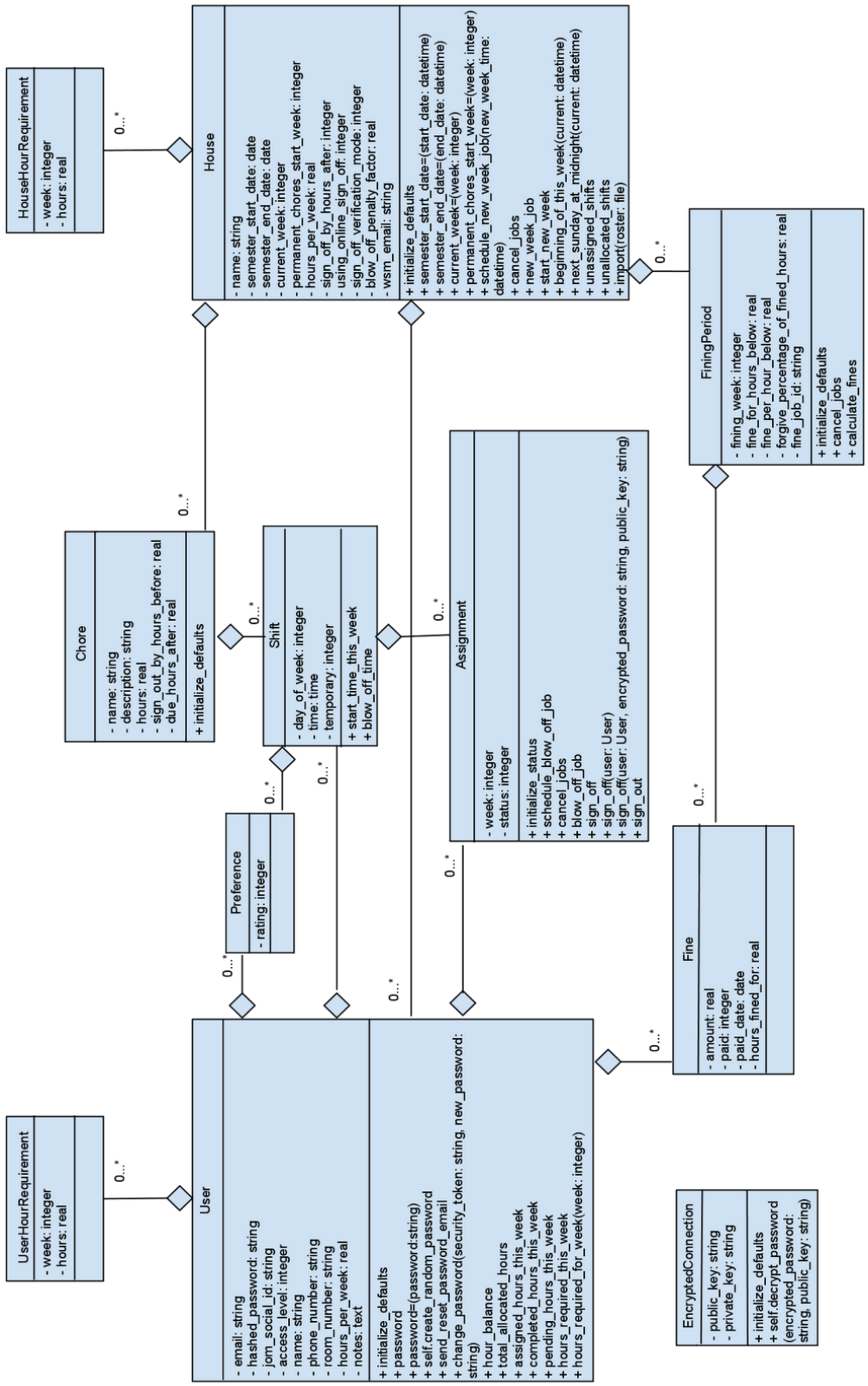
WSMA has one additional component, an asynchronous job scheduling system, Rufus-scheduler, which implements an event-based architecture. The model schedules jobs to be run at a specific time, and the scheduler invokes the model when that time arrives.

Design Details

ER Diagram


id
email
hashed_password
jom_social_id
access_level
name
phone_number
room_number
hours_per_week
notes
User
resident of
House
UserHourRequirement
week
HouseHourRequirement
id
hours
week_number
id
hours
id
name
semester_start_date
current_week
permanent_chores_start_week
hours_per_week
sign_off_by_hours_after
using_online_sign_off
sign_off_verification_mode
blow_off_penatly_factor
wsm_email
belongs to
Preference
id
rating
allocated
Assignment
id
week
status
Chore
instance of
Shift
id
day_of_week
time
termporary
belongs to
id
name
description
hours
sign_out_by_hours_before
due_hours_after
owes
Fine
pursuant to
FiningPeriod
id
fining_week
fine_for_hours_below
fine_per_hour_below
forgive_percentage_of_fined_hours
id
amount
paid
paid_date
hours_fined_for

Class Diagram



Class Descriptions

1. **User**
   Represents a user that will log into the WSMA, which can be a resident, a WSM, or an admin.

   Relations
   a. belongs_to **House** [must not be null]
   b. has_many **Fines**
   c. has_many **Shifts**
   d. has_many **Assignments**
   e. has_many **UserHourRequirements**

   Accessors
   f. **email: string** - email address [must not be null, must be unique].
   g. **password_hash: string** - an hashed version of the user's password using bcrypt [must not be null]
   h. **jom_social_id: string** - optional, for integration with JomSocial in the future
   i. **access_level: integer** - represents the user's privileges (1: resident. 2: WSM, can manage their own house. 3: admin, can manage all houses and create new houses) [must be 1, 2, or 3 and not null]
   j. **name: string** [must not be null]
   k. **phone_number: string**
   l. **room_number: string**
   m. **hours_per_week: real** - the number of hours per week this user is obligated to complete [must be >= 0 and not null]
   n. **notes: text** - notes the WSM wants to store about this user

   Methods
   o. **initialize_defaults** - called automatically by the after_initialize callback
      i. sets hours_per_week to this User's House.hours_per_week
   p. **password** - decrypts the password usng bcrypt and returns it
   q. **password=(new)** - encrypts the new password using bcrypt and stores it in password_hash
   r. **self.create_random_password** - creates a random 12-digit password of lowercase letters and numbers and returns it
   s. **send_reset_password_email** - sends an email to the user to change their password. NOTE: this is insecure if emails can be sniffed.
      i. creates a security token and places it in the Users table
      ii. stores the current time in the Users table
      iii. sends an email to the user's email address with a link to /Users/ change_password/user_id&st=[security_token]
   t. **change_password(security_token, new_password)** - changes the password to the new password given

    i. checks that the security token is correct
    ii. checks that the current time is less than 24 hours after the creation time of the security token
    iii. if both are ok, changes the password, deletes the security token from the database, and returns true
    iv. otherwise, returns false
u. **hour_balance** - Calculates the User's hour balance
    i. finds all Assignments to the current user which are completed, and sums their hour fields.
    ii. finds all Assignments to the current user which are blown off, and sums their hour fields. Multiplies this sum by the House.blow_off_penalty_factor and subtracts it from the total so far.
    iii. calls House.current_week to find the current week
    iv. for each week from 0 to (but not including) the current week, calls hours_required_week(week) on the current User, and subtracts the return value from the total so far.
    v. for all Fines associated with the User which have an associated FiningPeriod, add (Fine.hours_fined_for * FiningPeriod.forgive_percentage_of_fined_hours) to the total
    vi. returns the total
v. **total_allocated_hours** - Returns the number of hours of Shifts allocated to this User
    i. finds all Shifts allocated to the current user
    ii. sums their associated Chore's hour fields
    iii. returns the sum
w. **assigned_hours_this_week** - Returns the number of hours of Shifts Assigned to this User this week
    i. finds all Assignments for the current user for the current week
    ii. sums their associated Chore's hour fields
    iii. returns the sum
x. **completed_hours_this_week** - Returns the number of hours of Shifts completed by this User this week
    i. finds all Assignments for the current user for the current week with a status of completed
    ii. sums their associated Chore's hour fields
    iii. returns the sum
y. **pending_hours_this_week** - Returns the number of hours of Shifts pending for this User this week
    i. finds all Assignments for the current user for the current week with a status of pending
    ii. sums their associated Chore's hour fields
    iii. returns the sum
z. **hours_required_this_week** - Returns the number of hours this User is required to complete this week.
    i. finds the current week by calling House.current_week
    ii. returns hours_required_for_week(week number)

aa. **hours_required_for_week(week: integer)** - Returns the number of hours this User is required to complete for the given week. Must be given a non-null integer > 0.
    i. throws an exception if preconditions are not met
    ii. finds the UserHourRequirement for the current User and the given week, if it exists, and selects its hours field
    iii. finds the HouseHourRequirement for the House and the given week, if it exists, and selects its hours field
    iv. gets House.hours_per_week
    v. gets self.hours_per_week
    vi. returns the minimum of these four values, not considering any which are null

2. **House**
   Represents a BSC coop house, and holds its workshift settings.

   Relations
   a. has_many **Users**
   b. has_many **Chores**
   c. has_many **Shifts** through **Chores**
   d. has_many **FiningPeriods**
   e. has_many **HouseHourRequirements**

   Accessors
   f. **name: string** - Name of the house [must not be null, must be unique]
   g. **semester_start_date: datetime**. Date the semester starts. Week 1 starts the first Sunday following this. Reader only; writer overridden.
   h. **semester_end_date: datetime -** Date the semester ends. The system will not initiate any more weeks after this date. Reader only; writer overwritten.
   i. **current_week: integer** - The current week of the semester. Updated automatically by start_new_week. Reader only; writer overwritten. [Must be >= 0]
   j. **permanent_chores_start_week: integer** Week the permanent chores begin. This is the week to start automatically assigning shifts based on shift assignments. Reader only; writer overwritten. [Must be >=0 or null]
   k. **hours_per_week: real** - Number of hours all residents are required to complete per week [must be >= 0 and not null]
   l. **sign_off_by_hours_after: real** - The number of hours a resident has to sign off for their shift after the chore's due time [must be >=0 and not null]
   m. **using_online_sign_off: integer** - Whether or not the house is using online sign offs (0: false, 1: true) [must be 0 or 1 and not null]
   n. **sign_off_verification_mode: integer** - Determines whether a resident needs another resident to verify chore completeness during sign off, and if so, whether or not they need to authenticate to do so (0: no verification, 1: verification without authentication - emails are sent to confirm, 2: require authentication) [must be 0, 1, or 2 and not null]

o. **blow_off_penalty_factor: real** - The factor to multiply the number of hours a chore was worth if by when penalizing for blow offs (0 means no penalty, a blow off is equivalent to signing out, 1 means blowing off a 3-hour chore costs you 3 hours) [must be >=0 and not null]
p. **wsm_email: string**

Methods

q. **initialize_defaults** - called automatically by the after_initialize callback
    i. initializes hours_per_week, sign_off_by_hours_after, current_week, and blow_off_penalty_factor to 0 if they are not set yet
    ii. initializes using_online_sign_off to 1 if it is not set
    iii. initializes sign_off_verification_mode to 2 if it is not set
r. **semester_start_date=(start_date: datetime)** - sets the semester_start_date, and schedules the new week job. Given a datetime as a parameter, which must be valid and in the future. Must be called before the current semester_start_date, if it is set.
    i. throws an exception if preconditions are not met
    ii. sets semester_start_date to start_date
    iii. calls cancel_jobs
    iv. calls schedule_new_week_job with the first Sunday after the given date, at midnight as the argument
s. **semester_end_date=(end_date: datetime)** - sets the semester_end_date. Given a datetime as a parameter, which must be valid and in the future. Must be called before the current semester_end_date, if it is set.
    i. throws an exception if preconditions are not met
    ii. sets semester_end_date to end_date
t. **current_week=(week: integer)** - sets the current week. Given an integer as a parameter, which must be valid and after or equal to the current week.
    i. throws an exception if preconditions are not met
    ii. sets current_week to week
u. **permanent_chores_start_week=(week: integer)** - sets the week that permanent chores start. Given an integer as a parameter, which must be valid and after the current week.
    i. throws an exception if preconditions are not met
    ii. sets permanent_chores_start_week to week
v. **schedule_new_week_job(new_week_time: datetime)** - schedules a new week job to run at the given time
w. **cancel_jobs** - called automatically by the before_destroy callback
    i. cancels the Rufus-scheduler new week job
x. **new_week_job** - called by the new week job
    i. aborts if semester_end_date is not set or current date is past semester_end_date
    ii. calls start_new_week
    iii. if a week from now is before the semester_end_date, schedule next week's job
y. **start_new_week** - automatically assigns Shifts to the Users they are allocated to.

        i. increments current_week
        ii. if permanent_chores_start_week is set and current_week >= permanent_chores_start_week:
            1. for each Shift associated with this House with a non-null allocation field, create an Assignment between the associated User and the Shift
    z. **beginning_of_this_week(current: datetime)** - calculates and returns the sunday at midnight before the given datetime
    aa. **next_sunday_at_midnight(current: datetime)** - calculates and returns the sunday at midnight after the given datetime
    bb. **unassigned_shifts** - returns an array of shifts which are not assigned to anyone this week
        i. calls current_week to get the current week
        ii. queries for Shifts which have no corresponding Assignment for the current week
        iii. returns the results in an array
    cc. **unallocated_shifts** - returns an array of shifts which are not allocated to anyone and nontemporary
        i. queries for Shifts which have a user field value of nil and a temporary value of 0
        ii. returns the results in an array
    dd. **import(roster: file)** - Creates many users based on a list of residents. The input is a csv file containing the names and emails of the residents of the house.
        i. Iterates through each row in the file, instantiating a User for each one, with the given name and email address. For all Users, passes the House as an argument, and the hours_per_week of that House. For each User, creates a new random password.
        ii. Emails the default passwords to each created User, with a link to where they can change their password

3. **Chore**
   Represents a type of job which needs to get done in a House.

   Relations
    a. belongs_to **House** [must not be null]
    b. has_many **Shifts**

   Accessors
    c. **name: string** - name of the chore [must not be null]
    d. **description: string** - a description of the chore
    e. **hours: real** - the number of hours the chore takes to complete [must be >=0 and not null]
    f. **sign_out_by_hours_before: real** - the number of hours before the chore's assignment time a resident must sign out of the chore to be considered exempt from it. If the resident does not sign out before this time, he or she must complete

the chore or it will be considered blown off. [must be >= 0 and not null]

g. **due_hours_after: real** - how many hours after the chore's start time the chore is due. For many chores this will be equal to the hours field. For others (those without an assigned time to complete it) it may be much longer, for example 168 hours for a chore which starts at the beginning of the week and due at the end of the week. After this period, the resident has House.sign_off_by_hours_after hours to sign off for the shift. All chores are due by the end of the week, so for a Shift assigned at midnight of Saturday morning, a value of greater than 24 in the Chore it is based on will have no effect. [must be >= hours, <= 168, and not null]

Methods

h. **initialize_defaults** - called automatically by the after_initialize callback
   i. initializes sign_out_hours_before to 0
   ii. initializes due_hours_after to hours

4. **Shift**

Represents a work shift. Based on a Chore, but has an associated time, and can be allocated and assigned to a User.

Relations

a. belongs_to **Chore** [must not be null]
b. belongs_to **User**
c. has_many **Assignments**

Accessors

d. **day_of_week: integer** - the day of the week that the shift occurs on. (1: Sunday, 2: Monday, … 7: Saturday) [must be >=1, <=7, and not null]
e. **time: time** - the time the chore starts. Chore.sign_out_by_hours_before counts hours before [must not be null]
f. **temporary: integer** - whether or not this is a temporary shift (1: temporary, 0: not temporary). If a shift is marked as temporary it is intended to be assigned on a case by case basis, but never allocated, so it will not be displayed in the shift allocation interface. [must be 0 or 1 and not null] If temporary is 1, user must be nil

Methods

g. **start_time_this_week** - calculates the time this week the chore will start
h. **blow_off_time** - calculates the time this chore will be blown off if not yet signed off
   i. returns start_time_this_week + chore.hours + chore.due_hours_after + chore.sign_off_by_hours_after

5. **Assignment**

Represents the assignment of a Shift to a User. These will often be created automatically by House.start_new_week, but can be created manually when a User

signs into an available shift.

Relations

a. belongs_to **User** [must not be null]
b. belongs_to **Shift** [must not be null]
c. belongs_to **House** through User
d. belongs_to **Chore** through Shift

Accessors

e. **week: integer** - the week this Assignment pertains to [must be >= 0 and not null]
f. **status: integer** - the status of the Assignment (1: pending, 2: completed, 3: blown off) [must be 1, 2, or 3 and not null].

Methods

g. **initialize_status** - called automatically by the before_validation callback
    i. sets the assignment's status to 1 if the house is using online sign off, and 2 otherwise
h. **schedule_blow_off_job** - called automatically by the after_create callback
    i. if using online sign off, schedules blow off job for the shift's blow off time
i. **cancel_jobs** - called automatically by the before_destroy callback
    i. cancels the Rufus-scheduler blow off job
j. **blow_off_job** - called by the blow off job
    i. changes the assignment's status to 3
k. **sign_off** - Called when a User signs off for a shift they have completed. Not used when the WSM manually changes the status of a Shift. Should only be called when House.using_online_sign_off == 1 and House.sign_off_verification_mode == 0 on an Assignment which has a status of pending.
    i. throws an exception if preconditions are not met
    ii. checks the Assignment.status. if it is pending, changes it to completed
l. **sign_off(user: User)** - Called when a User signs off for a shift they have completed. Not used when the WSM manually changes the status of a Shift. Should only be called when House.using_online_sign_off == 1 and House.sign_off_verification_mode == 1 on an Assignment which has a status of pending.
    i. throws an exception if preconditions are not met
    ii. checks the Assignment.status. if it is pending, changes it to completed
    iii. sends an email to User notifying them that they verified for this Assignment
m. **sign_off(user: User, encrypted_password: string, public_key: string)** - Called when a User signs off for a shift they have completed. Not used when the WSM manually changes the status of a Shift. Should only be called when House.using_online_sign_off == 1 and House.sign_off_verification_mode == 2 on an Assignment which has a status of pending.
    i. throws an exception if preconditions are not met
    ii. checks the Assignment.status. if it is pending, calls User.authenticate(encrypted_password, public_key)
    iii. if it returned false, return false

        iv. if it returned true, changes Assignment.status to completed and return true
    n. **sign_out** - Called when a User signs out of a shift they cannot complete. Not used when the WSM manually unassigns a shift. Should only be called on an Assignment which has a status of pending.
        i. throws an exception if preconditions are not met
        ii. deletes this Assignment

6. **FiningPeriod**
Represents a scheduled fining period, and holds all the information about the fining policy.

Relationships

- a. belongs_to **House** [must not be null]
- b. has_many **Fines**

Accessors

- c. **fining_week: integer** - the week after which this fining period ends. The fine will be calculated and applied House.sign_off_by_hours_after hours after the end of this week. [must be >=0 and not null]
- d. **fine_for_hours_below: real** - The fining floor. Users will be fined only for any hours in their hour balance below -fine_for_hours_below. [must be <=0 and not null]
- e. **fine_per_hour_below: real** - Users will be fined $fine_per_hour_below for each hour their hour balance is below the fining floor. [must be >=0 and not null]
- f. **forgive_percentage_of_fined_hours: real** - At fining time, after a fine has been applied, forgive this percentage of a User's down hours (i.e. once a User has been fined for them, they count as if this percentage of them have been completed for the future). [must be >=0, <=1, and not null]
- g. **fine_job_id** - the id of the Rufus-scheduler job which applies the fine [must not be null]

Methods

- h. **initialize(house: House, fining_week: integer, fine_for_hours_below: real, fine_per_hour_below: real, forgive_percentage_of_fined_hours: real)** - example constructor usage with minimum parameters.
- i. **initialize_defaults** - called automatically by the after_initialize callback
    - i. creates a Rufus-scheduler job which will run calculate_fines at House.sign_off_by_hours_after hours after the end of fining_week
    - ii. stores the job id in the fine_job_id field
- j. **cancel_jobs** - called automatically by the before_destroy callback
    - i. cancels the Rufus-scheduler job with id fine_job_id
- k. **calculate_fines** - called by the Rufus-scheduler job after the end of the fining period, calculates and creates all fines

i. iterates though each User in the House, and for each User, calls hour_balance
ii. for any User with an hour balance less than fine_for_hours_below, create a fine associated with this FiningPeriod for the amount (fine_for_hours_below - hour_balance)*fine_per_hour_below, with a status of not paid. Store (fine_for_hours_below - hour_balance) in hours_fined_for

7. **Fine**
Represents a fine charged to a user. Usually created automatically by FiningPeriod.caclulate_fine, but can also be created manually by the WSM.

Relationships

a. belongs_to **User** [must not be null]
b. belongs_to **FiningPeriod**

Accessors

c. **amount: real** - the dollar amount of the fine [must be >0 and not null]
d. **paid: integer** - indicates of whether the fine has been paid yet (1) or not (0) [must be 0 or 1 and not null]
e. **paid_date: date -** the date on which the resident paid the fine, if paid [must not be null if paid is 1]
f. **hours_fined_for: real** - the number of hours the User was fined for [must not be null if associated with a FiningPeriod]

Methods

g. **initialize(user: User, amount: real, paid: integer)** - example constructor usage with minimum parameters.

8. **Preference**
Represents a User's preference rating for a Shift

Relationships

a. belongs_to **User** [must not be null]
b. belongs_to **Shift** [must not be null]

Accessors

c. **rating: integer** - how much the user prefers the Shift (0: unavailable, 1: strongly dislike, 2: dislike, 3: neutral, 4: like, 5: strongly like). [must be >=0, <=5, and not null]

Methods

d. **initialize(user: User, shift: Shift, rating: integer)** - example constructor usage with minimum parameters.

9. **UserHourRequirement**
   Represents the number of hours required for a User for a specific week, when it differs from the usual for that User.

   Relationships
   a. belongs_to **User** [must not be null]

   Accessors
   b. **week: integer -** the relevant week in the semester [must be >=0 and not null] Must be unique given user.
   c. **hours: real -** the number of hours required from the User for the week [must be >=0 and not null]

   Methods
   d. **initialize(user: User, week: integer, hours: real)** - example constructor usage with minimum parameters.

10. **HouseHourRequirement**
    Represents the number of hours required for a House for a specific week, when it differs from the usual for that House.

    Relationships
    a. belongs_to **House** [must not be null]

    Accessors
    b. **week: integer -** the relevant week in the semester [must be >=0 and not null] Must be unique given house.
    c. **hours: real -** the number of hours required from the User for the week [must be >=0 and not null]

    Methods
    d. **initialize(house: House, week: integer, hours: real)** - example constructor usage with minimum parameters.

View Details

The following are details on the different components and functionalities of each specific view of WSMA:

1. Constant View - The navigation sidebar on the site. It is always present. The components that are enabled for each user depend on the type of user, so non-logged-in users, residents, workshift managers (WSMs), and admins will all have different sets of components available to them. The “Enabled For” column in the table below makes note of who can see what.

Content	Component	Description	Enabled For
BSC Logo	Logo image	The logo of the Berkeley Student Cooperatives. Positioned at the top of the sidebar.	All
Site Title	Link	The title of the site, Workshift Manager (working title). Clicking on the title will take the user to their home page (view 3) if they are logged in, and will do nothing if they are not logged in.	All
Profile	Link	Link to the Profile page.	Residents, WSMs, admins
My Shifts	Link	Link to the My Shifts.	Residents, WSMs
Shift Preferences	Link	Link to the Preference page.	Residents, WSMs
My Fines	Link	Link to the Fines page.	Residents, WSMs
Allocate Shifts	Link	Link to the Workshift Allocation page.	WSMs
Monitor Shifts	Link	Link to the Monitor Shifts page.	WSMs
Chores	Link	Link to the Chores page.	WSMs
Manage Fines	Link	Link to the Manage Fines page.	WSMs
House Roster	Link	Link to the House Roster page.	WSMs, admins
Adminstrative Tools	Link	Link to the Administrative Tools page.	WSMs, admins

2. Log-in View - The home page for all users, the page everyone first sees when he/she enters the website. There are fields for a user to enter a user name and password. There is a button for new users that goes to a register view. There is also a submit button to submit information.

Content	Component	Descriptions
Sidebar	List	Constant View
User name	Dropdown box	The user selects his/her name from the dropdown box

Password	Text field	Each user has a unique password that will be input here and shows up as asterisks
Log-in	Button	Processes the user's log-in information.
New User?	Button	Takes the user to the Register View.
Forgot Password?	Button	Will be pressed if a user has forgotten his/her password. An email will be sent to the email of the specified user.

3. Register View - For new users, they need to enter their username and email address in order to receive the temporary password which they receive from the register page and log in with that password.

Content	Component	Description
Sidebar	List	Constant View
User name	Drop down box	The user selects his/her user name from the drop down box.
Email address	Text box	The user enters his/her email address which is stored in the central office.
Confirm	Button	Checks if the information is correct. If correct, displays the message asking the new user to check an email sent to that email address for the temporary password; if not, go back to register view.

4. Home View - The home page that users see right after they log in.

Content	Component	Description
Sidebar	List	Constant View
Wall of Fame/ Shame	Table	Shows the names of all the residents in the house along with their hour balance. If the resident has blown off shifts, it will show how many hours down they are, and if a resident has done extra shifts then it shows how many hours up they are.
Twitter Feed	Feed	A Twitter feed that shows recent messages from the bscannounce twitter account.

5. Profile View - The profile page is unique to each user.

Content	Component	Description

Sidebar	List	Constant View
Name	Text	Name of Resident
Password	Text	Password of Resident
New Password	Text field	New password the user would like to use. Shows up as asterisks
Confirm New Password	Text field	The same password as inputted in the New password text field.
Change Password	Button	Changes password to new password the user inputs. If Confirm New Password and New Password doesn't match, an error message will occur.
Room Number	Text	The room number for the current resident.
Email	Text	Current email of resident
New Email	Text Field	Email address resident would like to change to.
Change Email	Button	Changes email to email in New Email field. Error will occur if not viable email.
Phone Number	Text	Current phone number for resident
New Phone number	Text field	New number resident would like to change to
Change Phone Number	Button	The phone number in the New Phone Number field will now be the current phone number.

6. Administrative Tools page - This page allows the WSM and admin to edit certain house parameters and to make backups of the database. Each content in this view can be changed

Content	Component	Description
Sidebar	List	Constant view.
Semester Start	Text Field	Date that the semester started on
Semester End	Text Field	End date for the semester

Hours per week	Text Field	Number of hours all residents are required to complete per week
Sign off by hours after	Text Field	The number of hours after a resident has to sign off for their shift after the chore's due time
Using online sign off	Check Box	Whether or not the house is using online sign offs
Sign off verification mode	Drop down menu	Determines whether a resident needs another resident to verify chore completeness and if authentication is needed
WSM email	Text Field	Email of the WSM
Back Up	Button	Backs up the database for the house.

7. Chores Page -This page is only viewable to the Workshift manager. Chore creation and editing will happen here.

Content	Component	Description
Sidebar	List	Constant View
Name	Text field	- The name of a chore
Description	Text field	- Description of a chore
Time	Text field	- A time interval in which a chore will be preformed.
Day	Drop down menu	- Selection between the days of the week.
Add Shift	Button	- Adds the shift to the form for the chore
Number of Hours	Text field	- The number of hours a chore will take
Sign Out Deadline	Text field	- How far in advance (in number of hours) a person needs to sign out of a task
Created Chores	List	-A list of currently created Chores - Clicking on an element in this list will produce a pop-up:

Content	Component	Description
Description	Text	-Description of chore

		<table><tr><td>Edit</td><td>Button</td><td>- Bring the Chore View[5] into focus with the Chore information loaded.</td></tr><tr><td>Delete</td><td>Button</td><td>- Delete the chore</td></tr></table>

8. Preference Page - This page is only viewable to the residents, allowing them to edit and sumbit their preferences on each workshift they want to do.

Content	Component	Description
Sidebar	List	Constant View
Explanation	Link	Produces a pop-up window that has the following components: <table><tr><td>Content</td><td>Component</td><td>Description</td></tr><tr><td>Details</td><td>Text</td><td>Details on what each work shift is about.</td></tr></table>
Rank	Table	**See Figure 1.1**
Submit	Button	Submits the workshift preference information

9. House Roster Page - This page is only accessible by the Workshift Manager. It is unique for each Co-op.

Content	Component	Description
Sidebar	List	Constant View
Roster	Table	**See Figure 1.2**
Edit	Button	Allows the WSM to edit the information of each residents <table><tr><td>Content</td><td>Component</td><td>Description</td></tr><tr><td>Name</td><td>Textfield</td><td>Textfield to edit the name</td></tr><tr><td>Email address</td><td>Textfield</td><td>Textfield to edit the email address</td></tr></table>

		<table><tr><td>Room Number</td><td>Textfield</td><td>Textfield to edit the room number</td></tr><tr><td>Submit</td><td>Button</td><td>Submits the modified information</td></tr></table>

10. Workshift Allocation Page - This is the page where WSMs allocate shifts to residents.

Content	Component	Description
Sidebar	List	Constant View

<table>
<tr><td>Daily Chores</td><td>Tab</td><td>Shows all shifts of chores that have to be done every day. Allows the WSM to see residents' preferences for shifts and to allocate the shifts to residents.

<table>
<tr><td>Content</td><td>Component</td><td>Description</td></tr>
<tr><td>Chores</td><td>Column</td><td>Lists all the daily chores with their time slots and number of hours.</td></tr>
<tr><td>Assignments</td><td>Calendar</td><td>For each chore, there is a row in the calendar.</td></tr>
<tr><td>Add</td><td>Button</td><td>Produces a pop-up window that has the following components:

<table>
<tr><td>Content</td><td>Component</td><td>Description</td></tr>
<tr><td>Name</td><td>Textfield</td><td>Input the name of the special workshift</td></tr>
<tr><td>Resident</td><td>Textfield</td><td>Input the name of the resident doing the shift</td></tr>
<tr><td>Submit</td><td>Button</td><td>Submit the information</td></tr>
</table></td></tr>
<tr><td>Submit</td><td>Button</td><td>Submit the information</td></tr>
</table></td></tr>
<tr><td>Weekly Chores</td><td>Tab</td><td>- Switches allocation matrix to Weekly Chores view. See figure 1.7</td></tr>
<tr><td>Resident List</td><td>Sidebar</td><td>- A list of residents
- When a residents name is clicked, their top rated chores are highlighted.</td></tr>
</table>

		- The current number of hours a resident has been assigned will be visible. - A resident name can be dragged and dropped into shift slots in the shift allocation matrix.
“x”	Button	- A small button on in the top right corner of a resident name in the shift matrix. - This button removes the resident from the shift slot and updates the number of hours currently allocated to the resident.

11. Monitor workshifts Page - This page allows the WSMs to view the shifts of the week, and add special one time shifts for the week.

Content	Component	Description
Sidebar	List	Constant View
Shifts of the week	Table	**See Figure 1.3**
Edt	Button	Allows the WSM to edit the details of each allocated workshift.

Content	Component	Description
Resident	Textfield	Edit the person who is assigned to a specific workshift.
Add	Button	Produces a pop-up window that has the following components:

Content	Compnent	Description
Name	Textfield	Input the name of the special workshift
Resident	Textfield	Input the name of the resident doing the shift
Submit	Button	Submit the information

		<table><tr><td></td><td></td><td></td></tr><tr><td>Submit</td><td>Button</td><td>Submit the information</td></tr></table>
Weekly	Tab	- Switches the shifts table to a table of week long shifts.
Daily	Tab	- Switches the shifts table to a calendar view
Create Shift	Button	- Produces a slide out panel that has the same content and components of the Chore View[5]

12. My Shifts Page - This page is where the residents manipulates their workshifts: sign up, sign off or take shifts that are available.

Content	Component	Description
Sidebar	List	Constant View
Sign off	Button	Allows the user to sign off a workshift, changing the status of that specific workshift to be completed
Sign out	Button	Allows the user to sign out of a workshift, changing the status of that specific workshift to be available.
Change view	Button	Changes the display of that current week's workshifts to current month's
My shift	Table	**See figure 1.4**
Show available shifts	Button	Displays the available shifts that were signed out by other residents

13. Manage Fining - This page is only viewable to the WSM where he/she can set up the fining period, the rate on how much to fine, and view the table on each resident's fine.

Content	Component	Description
Sidebar	List	Constant View
Current Fine	Table	**See Figure 1.5**
Current Fining Period	Text	Displays the current fining period

Current rate	Text	Displays the current rate
Edit	Button	Produces a pop-up window that has the following components: Content / Component / Description New Fining Period / Text field / WSM enters the new fining period New rate / Text field / WSM enters the new rate for fine Submit / Button / Submits the new data

14. Fines - This page is only viewable to the residents where they can view their net hours and the fine they owe

Content	Component	Description
Sidebar	List	Constant View
Current Fine	Table	**See table 1.6**

Figure 1.1

Name	Preference (0: absolutely no; 5: definitetly yes)
Post-dinner pots	o 0 o 1 o 2 o 3 o 4 o 5
Post-dinner dishes	o 0 o 1 o 2 o 3 o 4 o 5
Dishes before 3pm	o 0 o 1 o 2 o 3 o 4 o 5
Dinner clean	o 0 o 1 o 2 o 3 o 4 o 5

Figure 1.2

Name	Email address	Room Number
Kylie	kylie@yahoo.com	#321

Adele	adele@aol.com	#123
Mary	mary@hotmail.com	#231

Figure 1.3

Date	Workshift	Resident	Status
Monday	Post-dinner pot	Mike	Completed
	Post-dinner dishes	Kenny	Pending
Tuesday	Post-dinner dishes	Ralph	Blown off
	Post-dinner pot	Brian	Available

Figure 1.4

Tuesday	Wednesday	Thursday
Post-dinner pots	Pots before noon	Dishes from 9pm-11pm
Dishes after 10pm		Late night pots

Figure 1.5

Name	Net Hours	Fine
Travis	2.5	$0
Jimmy	-5.0	$50

Figure 1.6

Week	Net Hours	Fine
0	0.0	$0
1	-5.0	$50
2	-2.5	$0

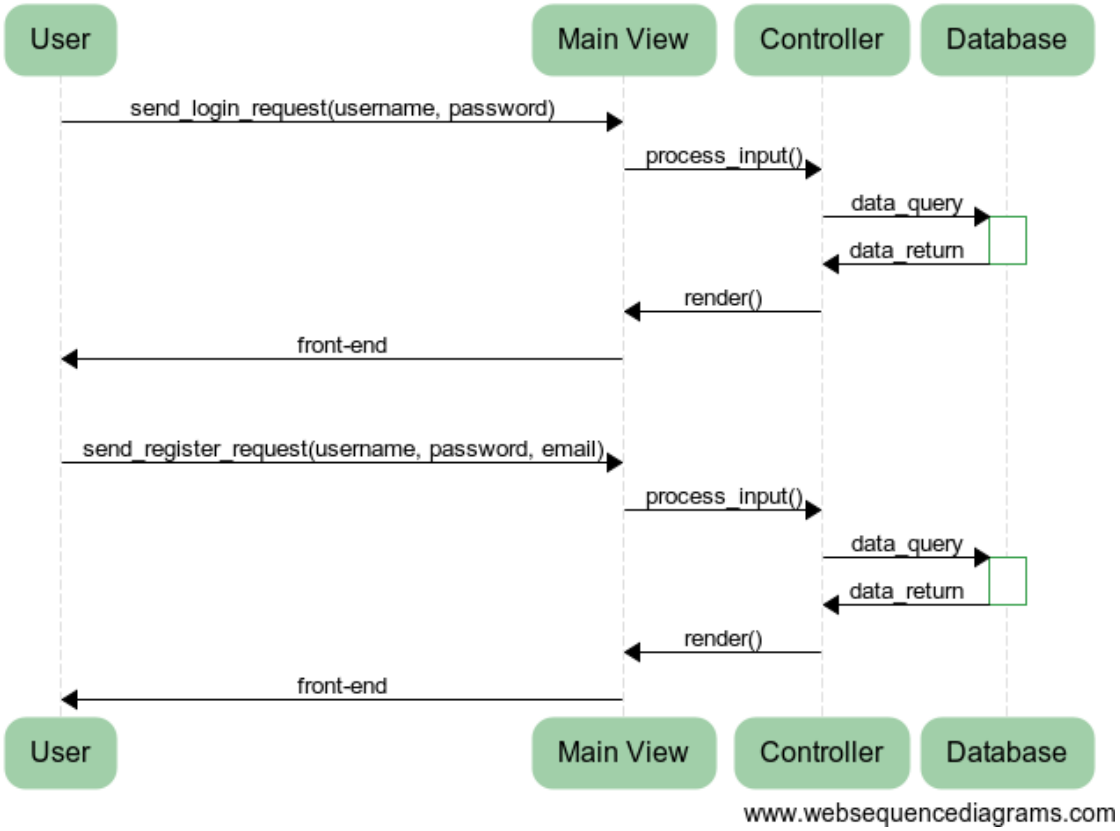
3	0.0	$0

Figure 1.7

Chore	**Resident**(s)
Clean Bathroom	Billy / Bob
Clean Kitchen	Yoko / Ono / Bjork

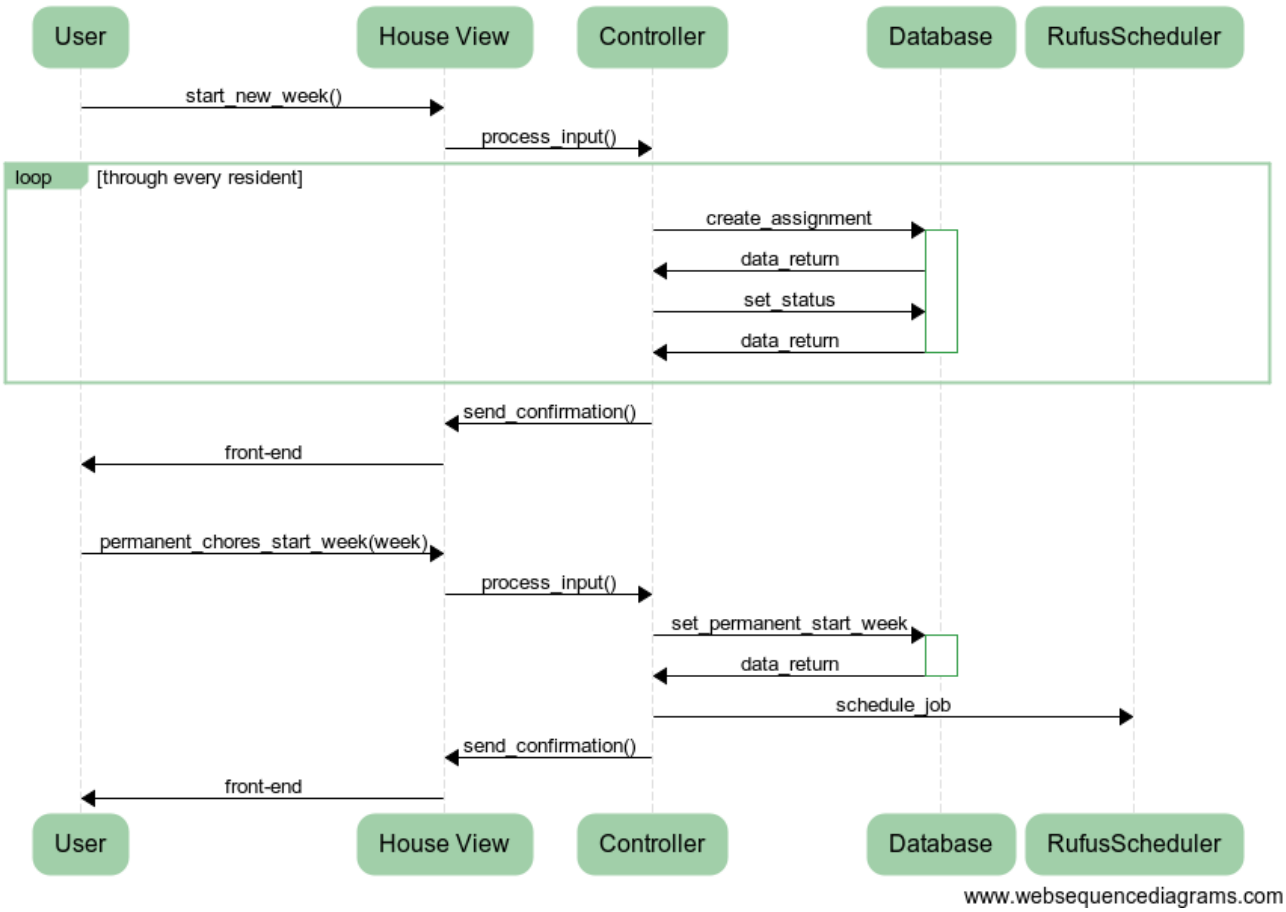
Sequence Diagrams

Login View



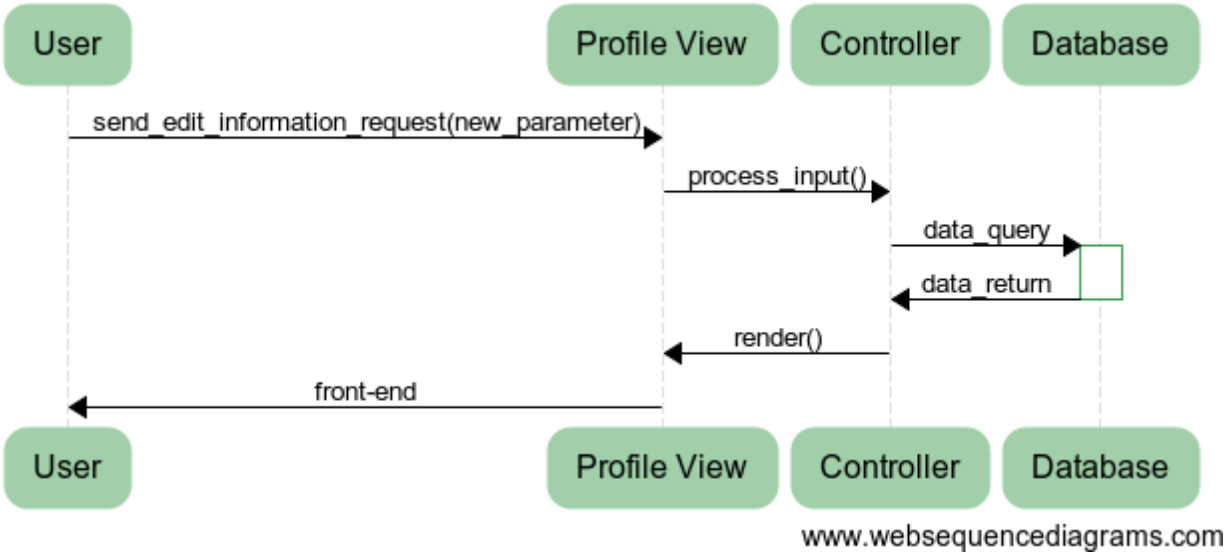
The user can login or register on this page. the controller will verify the login information with the database to check for errors, and will notify the user if any information is incorrect.

House View



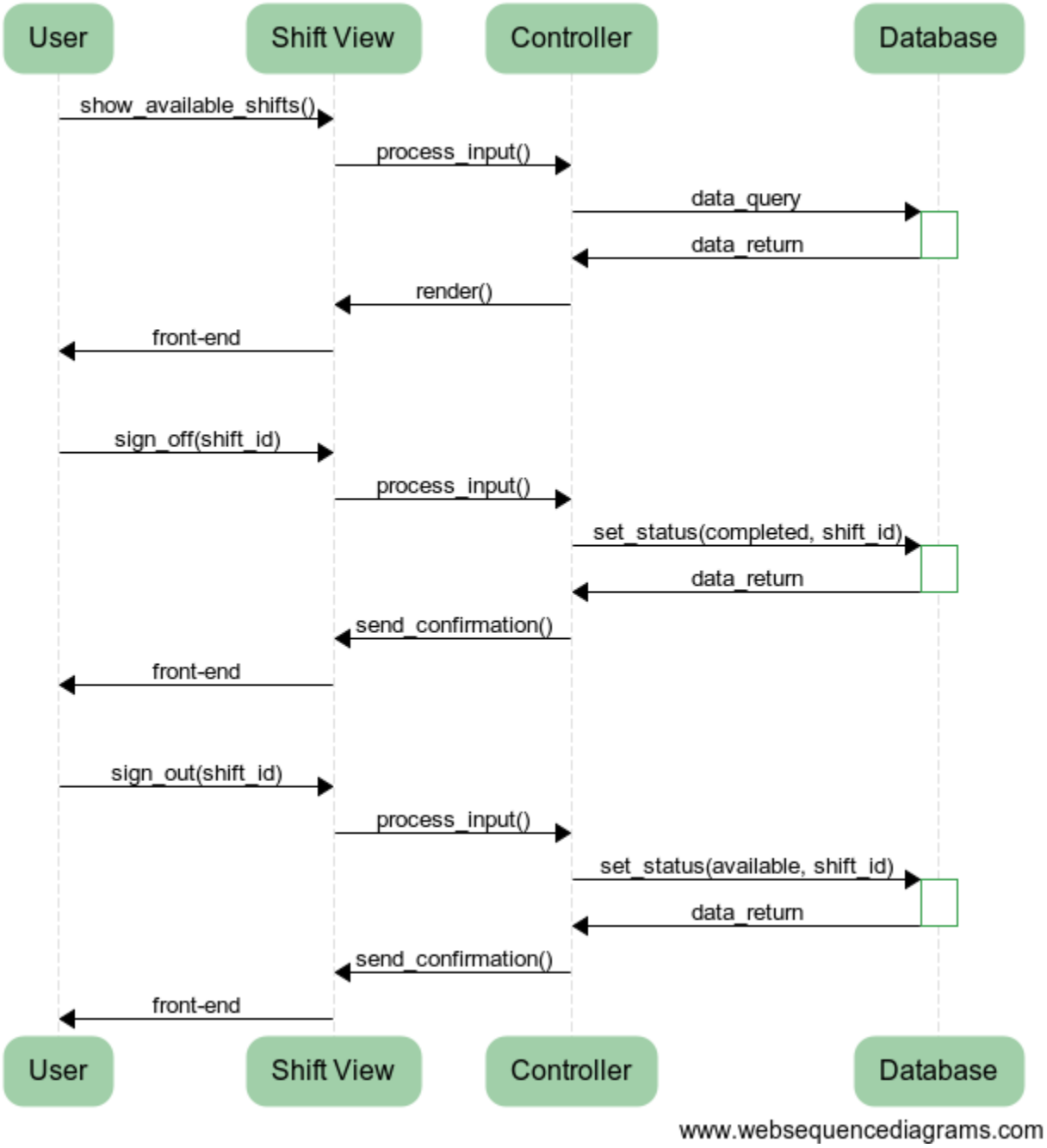
The house view contains methods needed to assign shifts and will likely be invoked every week by the workshift manager. The start_new_week() method loops through every resident and assigns the shift it is allocated to. It also marks the status of each shift as pending.

Profile View



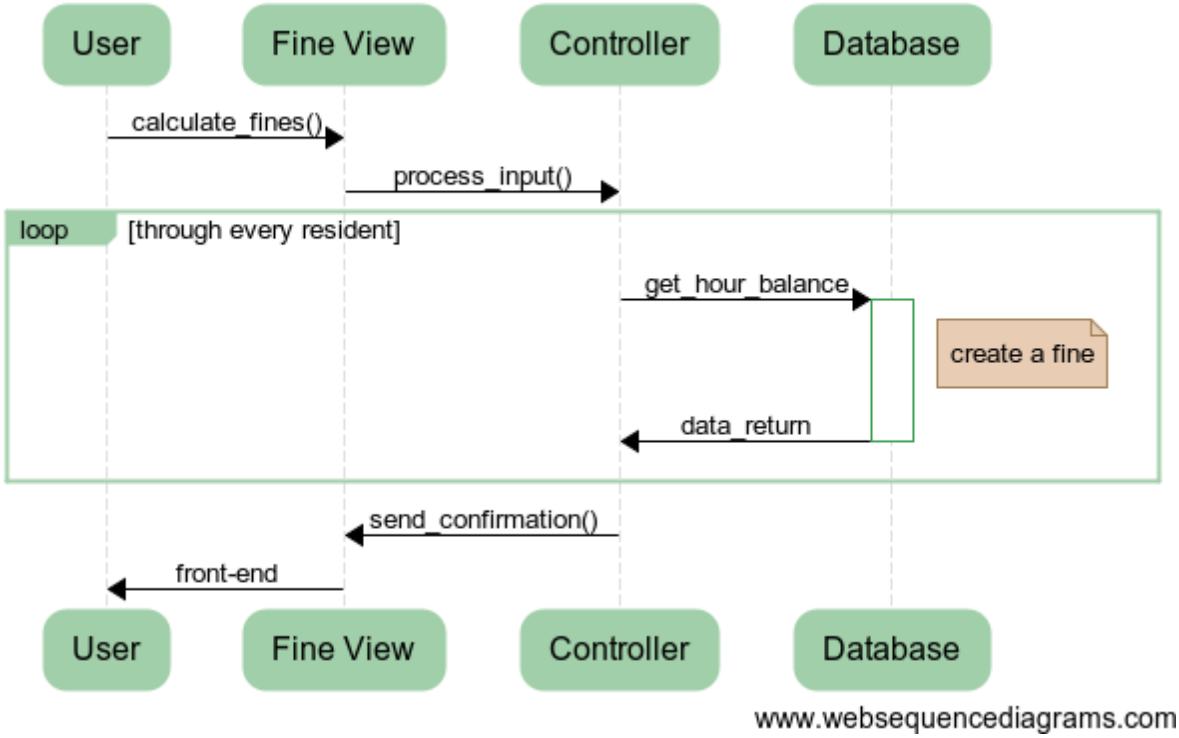
The profile view is a simple page that allows residents to see the details of their profile. The information can be easily edited anytime through this page.

Shift View



The shifts page allows residents to see the available shifts, and they can also sign-off or sign-out of their assigned shifts.

Fine View



The fine view allows the workshift manager to calculate the fines for an entire house, according to the fining period that they set before.

Authentication

Authentication is done using the Rails Session, which contains built-in security features, and bcrypt.

1. User requests the login method of the Users controller
2. A user signs in with their email and password with a POST to the login method of the Users controller
3. The Users controller checks that the password is correct using bcrypt (the plaintext passwords are not stored in the database).
4. The Users controller stores their user id in the Session
5. To authenticate before seeing a page, include a before_filter on the respective controller method with a call to authenticate, authorize_admin, authorize_wsm or authorize_user as appropriate.
6. The authenticate and authorize methods, defined in ApplicationController, check that a user is logged in (via the session) and checks their privileges. If the priviliges are insufficient, it redirects to the login page.

If a user forgets their password, they can reset it.

1. User requests the forgot_password method of the Users controller
2. The user enters their email address and POSTs it to the forgot_password method of the Users controller

3. The Users controller finds the User by email and calls the send_password_reset_email method of the User model.
4. The user checks their email and clicks on the included link, which requests the change_password method of the Users controller, and passes a security token.
5. The Users controller preloads the security token into the form.
6. The user enters their new password and POSTs to the change_password method of the Users controller.
7. The Users controller calls the change_password method of the User model, and responds to the user based on the success of the operation.

Anticipated Problems

- Integration of the html view mockups into the Rails MVC framework
- Difficulty setting up, configuring, or using Rails, since we are all new to it
- Learning how to use the version control system
- Learning how to use jquery
- Additional features that may be added later on
- Difficulty scheduling meetings with the users for usability studies
- Developing an intuitive UI that the customer will like
- Compatibility between different parts of the system

Limitations

- Since we are designing the WSMA for a specific customer, it may have limitations insofar as general usage. It may not scale to larger production environments, since it is designed with only the size of the BSC houses in mind. The user interface will be catered to the needs, abilities, and aesthetics of a college student, but may not be as intuitive for other users. The features provided and not provided are also chosen specifically based on the use cases of the BSC, other users may have different needs.

Alternative Designs

- As a possible alternative to writing our own authentication system, we could use an existing framework (like devise). However, our needs are relatively simple, so we wanted to roll our own simple authentication system to avoid the overhead and complexity of using another system. Doing so also allows us to seamlessly integrate the authentication system with the User model, and therefore directly with the concept of a resident or WSM.

Implementation plan

Since the WSMA is a Rails web application, we will attempt an agile development process. We will have three iterations, each taking approximately two weeks to complete. We will split the CS 169 course Iteration 1 in half to create our team's Iteration 1 and 2, and then the CS 169 course Iteration 2 will align with our Iteration 3. Henceforth in this section iteration numbering will refer to our team's iteration schedule, not the CS 169 course iteration schedule. Our team will be split

into a 3-person frontend team and a 2-person backend team.

- 10/7 - 10/21: For the first iteration, we will aim to have a working prototype to show our users. This will involve interactive html/javascript prototypes of the views. By the end of the first iteration, we also aim to have a working boilerplate and scaffold for the entire model. The mechanics of the rails architecture should be configured, and user authentication should be implemented.
- 10/21 - 11/4: For the second iteration, we will aim to have a working prototype of the WSMA's central functionality. This requires an actualization of the frontend in terms of Rails views instead of pure html, an implementation of much of the backend, and integration between the frontend and backend.
- 11/4 - 11/18: In the third iteration, we will finish the full implementation of the backend and frontend functionality and fully integrate them.

Time estimates for features (in hours)

1. Overall layout - 10
2. Authentication - 10
3. Admin Manage Houses - 4
4. House roster - 6
5. WSM manage users - 7
6. User manage settings - 4
7. WSM manage chores and shifts - 16
8. WSM allocate shifts - 20
9. WSM manage shift assignments - 8
10. User manage assigned shifts - 8
11. User sign off for a workshift - 8
12. Import list of residents - 8
13. Preference form - 6
14. WSM manage fining periods - 12
15. WSM manage fines - 4
16. User manage fines - 2

Delegation of Tasks

Views:
Login - Tai
Allocation - Cat
My Shift -Travis
Manage Fines - Travis
Register - Travis
Profile - Travis
Home Page - Cat
Chores - Tai
Monitor Shifts - Tai
Administrative Tools - Cat
Fines (Resident) - Tai
Preferences - Cat

Roster - Travis
Constant View (Sidebar) - Cat

1. HouseRoster:
    a. Profile view->House roster view
    b. No functions in class will be called
    c. Contents involved:
        i. view the roster (button)
        ii. edit the roster (button)
        iii. create new users (pop-up window)
    d. Effort: easy, since these features are all provided through rails.
2. Create Chore:
    a. Workshift Allocation Page view:
        i. Daily Chores:
            1. Add:
                a. Name (Text)
                b. Residents (Text)
                c. Submit (Button)
    b. initialize(house: House, name: string, hours: real) in the Chores class will be used to create the chore.
    c. Effort: medium, as despite the features provided by rails, back-end implementation will be done in order to achieve creating the chores.
3. Create Shift
    a. Monitor workshifts page view:
        i. Create shifts:
            1. A slide out panel
    b. initialize(chore: Chore, day_of_week: integer, time: time, temporary: integer) method in the shift class will be used.
    c. Difficulty: medium, despite the javascript that's going to generate the slide-out panel, back-end implementation needs to support the initialize method.
4. Allocate Shift
    a. Workshift Allocation Page view:
        i. Daily Chores:
            1. Chores (column)
            2. Assignments (calendar)
            3. Add (button)
            4. Submit (button)
    b. The initialize method will be used to create new shifts
    c. Effort: Easy/Medium. The columns, calendars and buttons can be done in our iteration 1, but the allocating shifts has to be done in iteration 2.

5. Sign out of Shift
    a. My shift page view:
        i. Sign out of a workshift (button)
    b. Monitor Workshifts page view:

    i. Shifts of the week (table)
c. sign_out() from class assignment will be used to indicate which specific resident signs out of a work shift.
d. Effort: Easy Medium. The buttons and tables can be done in iteration 1, but the signing out of a workshift functionality will need to be implemented in iteration 2 as it involves interaction with the back end.

6. Sign off a workshift
    a. My shift page view:
        i. If using online sign-offs,
            1. Sign off of a workshift (button) - Iteration 1
        ii. Else
            1. No action required (WSM will manage)
    b. Monitor Workshifts page view:
        i. Shifts of the week (table) will be updated as respective shift is set to completed.
    c. sign_off() from class Assignment will be used to indicate which specific resident signs off of a work shift. - Iteration 2
    d. Effort: Easy Medium. The buttons and tables can be done in iteration 1, but the signing off of a workshift functionality will need to be implemented in iteration 2 as it involves interaction with the back end.
7. Import list of residents
    a. Profile view->House roster view
        i. Import list button - Iteration 1
    b. Parse list, extract names and other info - Iteration 2
    c. Call add_user() function for each user in list - Iteration 1
    d. Effort: Medium. Should be implemented in 2nd or 3rd iteration, needs add single user functionality first.
8. Create a house
    a. Administrative tools page
        i. Semester Start
        ii. Semester End
        iii. Hours per week
        iv. Sign off by hours after
        v. Using online sign off
        vi. Sign off verification mode
        vii. WSM email
    b. Each field will be editable, database should be updated to reflect any changes.
    c. Effort: Easy, Ruby provides functionality for editing fields. Should be implemented in iteration 1.
9. Residents fill in/update the workshift preference form
    a. Preference page
        i. Preference form (table) - Iteration 1
        ii. Submit button - Iteration 1
    b. Preference values for resident is saved into database - Iteration 2
    c. Effort: Easy Medium. User interface (preference table) can be implemented in iteration 1, but the connection to the database can be left to iteration 2.
10. Add people to roster
    a. Profile view->House roster view
        i. Add resident button - Iteration 1
    b. Call add_user() function using information submitted - Iteration 1

c. Effort: Easy. Should be implemented in 1st iteration, needed for other functions like import list of residents.

11. Change Default Password
    a. Create Profile view -- 4 units -- iteration 2
    b. Create User class -- 5 units -- iteration 1
    c. Create EncryptedConnection class -- 2 units -- iteration 3
    d. set_password function in User class needs to be called -- 1 units -- Iteration 1
    e. make sure New Password Field and Change Password field are the same -- 1 unit -- iteration 2
12. Backup System
    a. Administration Tools page -- 2 units -- Iteration 3
    b. All databases with relevant information to the specific house needs to be backed up to a csv file -- 3 units -- iteration 3
13. Change the number of hours a resident owes
    a. House Roster view -- 2 units -- iteration 2
        i. click on resident name, edit number of hours
    b. set_hours_per_week function -- 1 unit -- iteration 2
14. Set the fining period
    a. Manage Fining view -- 2 units -- iteration 2
    b. initialize **-** method in the Fining Period class needs to be called -- 1 unit -- iteration 2.
    c. Rufus Scheduler needs to be configured to run at the end of every fining period -- 3 units -- iteration 3.
15. View the history of one's total workshifts done
    a. My Shift Page -- 2 units -- iteration 1
        i. List of previous Workshifts and their status
    b. Each time a shift is signed out, signed off, or blown off, a list on the My Shift Page view needs to be appended along with the appropriate database entries. -- 2 units -- iteration 2
16. Set default fine amount
    a. Create the User class -- 1 unit -- Iteration 1
    b. Create the House class -- 1 unit -- Iteration 1
    c. Create the FiningPeriod class -- 1 unit -- Iteration 1
    d. Make the log-in page -- 1 unit -- Iteration 1
    e. Write log-in script that looks up username and password, checks that they are a valid pair, checks what kind of user this is, and redirects the user to their home page -- 4 units -- Iteration 1
    f. Make the WSM Home page -- 2 units -- Iteration 2
    g. Make the Manage Fining page -- 2.5 units -- Iteration 2
    h. Write script for sending the default fine amount data to the backend and updating the relevant FiningPeriod -- 4 units -- Iteration 2
17. Manually fine a resident
    a. Create User class -- 1 unit -- Iteration 1
    b. Create Fine class -- 1 unit -- Iteration 1
    c. Make the log-in page -- 1 unit -- Iteration 1
    d. Create the log-in script that looks up username and password, checks that they

are a valid pair, checks what kind of user this is, and redirects the user to their home page -- 4 units -- Iteration 1

e. Make the WSM Home page -- 2 units -- Iteration 2
f. Make the Manage Fining page -- 2.5 units -- Iteration 2
g. Write script for sending fine data to backend and updating the relevant Fine -- 4 units -- Iteration 2

18. Create special shift
    a. Create User class -- 1 unit -- Iteration 1
    b. Create House class -- 1 unit -- Iteration 1
    c. Create Shift class -- 1 unit -- Iteration 1
    d. Create Assignment class -- 1 unit -- Iteration 1
    e. Make the log-in page -- 1 unit -- Iteration 1
    f. Create the log-in script that looks up username and password, checks that they are a valid pair, checks what kind of user this is, and redirects the user to their home page -- 4 units -- Iteration 1
    g. Make the WSM Home page -- 2 units -- Iteration 2
    h. Make the Monitor Workshifts page with the slide-out panel with the create shift form -- 3 units -- Iteration 2
    i. Write script that sends create shift form data to backend and creates a new Shift and a new Assignment -- 4 units -- Iteration 2
19. Manually sign off a resident
    a. Create User class -- 1 unit -- Iteration 1
    b. Create House class -- 1 unit -- Iteration 1
    c. Create Shift class -- 1 unit -- Iteration 1
    d. Create Assignment class -- 1 unit -- Iteration 1
    e. Make the log-in page -- 1 unit -- Iteration 1
    f. Create the log-in script that looks up username and password, checks that they are a valid pair, checks what kind of user this is, and redirects the user to their home page -- 4 units -- Iteration 1
    g. Make the manual sign off tab in the Monitor Shifts page -- 4 units -- Iteration 2
    h. Write script that sends sign-off data to backend and updates the relevant Assignments -- 4 units -- Iteration 2
20. User log in
    a. Create User class -- 1 unit -- Iteration 1
    b. Make the log-in page -- 1 unit -- Iteration 1
    c. Create the log-in script that looks up username and password, checks that they are a valid pair, checks what kind of user this is, and redirects the user to their home page -- 4 units -- Iteration 1

Iteration 1

HTML mockups:

- Shift Allocation - Cat
- Preferences - Cat
- Chores - Tai
- Roster - Travis
- Sidebar - Cat

View:

- Login - Tai
- Register - Travis

Controller:

- Login - Tai
- Register - Travis

Model:

- ~~User Class needs to be fully implemented - Brian~~
- Create validation testing framework - Brian
- Scaffolding, validation and validation unit testing:
    - User - Brian
    - House - Brian
    - EncryptedConnection - Brian
    - HouseHourRequirement - Ben
    - UserHourRequirement - Ben
    - Preference - Ben
    - Fine - Ben
    - FiningPeriod - Ben
    - Assignment - Ben
    - Shift - Ben
    - Chore - Ben
- Link User and House models together - Brian
- Link UserHouseRequirement to User and HouseHourRequirement to House - Brian

Weekly plan

- Week 1
    - Backend
        - Set up repository and rails project - Brian
        - Create scaffold for all model classes - Brian and Ben
        - Create validation testing framework - Brian
    - Frontend
        - Create HTML mockups for Shift Allocation, Preferences, Chores, Roster, Sidebar, Login, Register
- Week 2
    - Backend
        - Implement validation for the User, House, and EncryptedConnection classes, and tests for that validation - Brian
        - Link User and House models together - Brian
        - Link UserHouseRequirement to User and HouseHourRequirement to House - Brian
        - Implement validation for the rest of the model classes, and tests for that validation - Ben
    - Frontend
        - Implement jQuery effects on existing HTML mockups
        - Controllers and Views for Login and Register

Iteration 2

During this iteration, the web pages developed during the last iteration needs to be integrated

into Rail Views. The rest of the pages should be mocked up in html.
HTML Mockup:

- My Shift - Travis
- Manage Fines - Travis
- Profile - Travis
- Home Page - Cat
- Monitor Shifts - Tai
- Administrative Tools - Cat
- Fines (Resident) - Tai

View:

- Shift Allocation - Cat
- Preferences - Cat
- Chores - Tai
- Roster - Travis
- Sidebar - Cat

Controller:

- Shift Allocation - Cat
- Preferences - Cat
- Chores - Tai
- Roster - Travis

Model:

- User - Brian
- House - Brian
- Chore - Ben
- Shift - Ben
- Assignment - Ben
- Preference - Ben

Weekly plan

- Week 3
    - Backend
        - Implement login/authentication system - Brian
        - Implement all functionality of Chore, Assignment, Shift, and Preference - Ben
    - Frontend
        - Install application layout - Cat
        - Create Views for Fines and Monitor Shifts - Tai
        - Controllers for Chores page and Fines - Tai
        - Implement jQuery draggable effect on Shift Allocation mockup - Cat
- Week 4
    - Backend
        - Implement all functionality of User and House except import, jobs, and advanced auth features - Brian
        - Implement all functionality of Fine, FiningPeriod, HouseHourRequirement, UserHourRequirement - Ben

    - Frontend
        - Make views from all remaining mockups - Team
        - Controllers for any other completed views - Team
        - Identify and implement any jQuery effects on pages beside Shift Allocation - Team
        - Decide once and for all on color scheme, fonts, and other decoration details - Team
        - Do user testing - Team

Iteration 3

At this point, most of the functionality will be done. Pages from Iteration 2 that were mocked up in html will need to be integrated into Rail Views with controllers for those pages. Extra features such as backing up the database will be done here if there is time. Implementing encryption for logins and passwords will be done in this step.

- Week 5-6
    - Backend
        - Implement scheduled jobs, import, and advanced auth features - Brian
        - Create administrative interface - Brian

Testing plan

Testing is Rails is easy, because a testing framework is built right into the scaffolding created when a new project is created. Rails is designed for agile development, which stresses a test-driven development cycle with rapid iterations. This agility will be useful when creating the WSMA, because it will allow us to show something to our users as soon as possible, and to involve them in the development process. Even before writing code, we will write ruby tests which represent acceptance tests given during the specification phase of development. They allow effective testing of the model validation and business logic without needing to work through the views, as one would during manual testing. They also allow for built-in regression testing. Whenever making changes to the software, we can re-run our test suite to make sure nothing broke.

Unit tests in Rails test a specific class in the model. In the first iteration, we will write unit tests to make sure that each class's fields are being validated properly. In the second and third iterations, we will add unit tests for the business logic of the model as well, including passing invalid parameters to functions and executing corner cases. Fixtures can be used to simulate existing database state when running tests in Rails, so we can easily test how the logic will work on realistic sample data.

Rails functional tests are used to verify that controllers behave as expected. The testing framework can simulate http requests and inspect the responses. Integration tests can test

an entire user story step by step, tracing an interaction across multiple requests to different controllers. Functional tests will be written along with controller code starting in the first iteration, but mostly in the second and third. Integration tests will be written in the second and third iterations.

System testing will be done manually both by us and by our users. System testing can find issues automated tests cannot, such as usability problems, missing features, and browser compatibility issues. This type of testing will be done throughout the entire development process, from our interactive UI prototypes to the finished product.