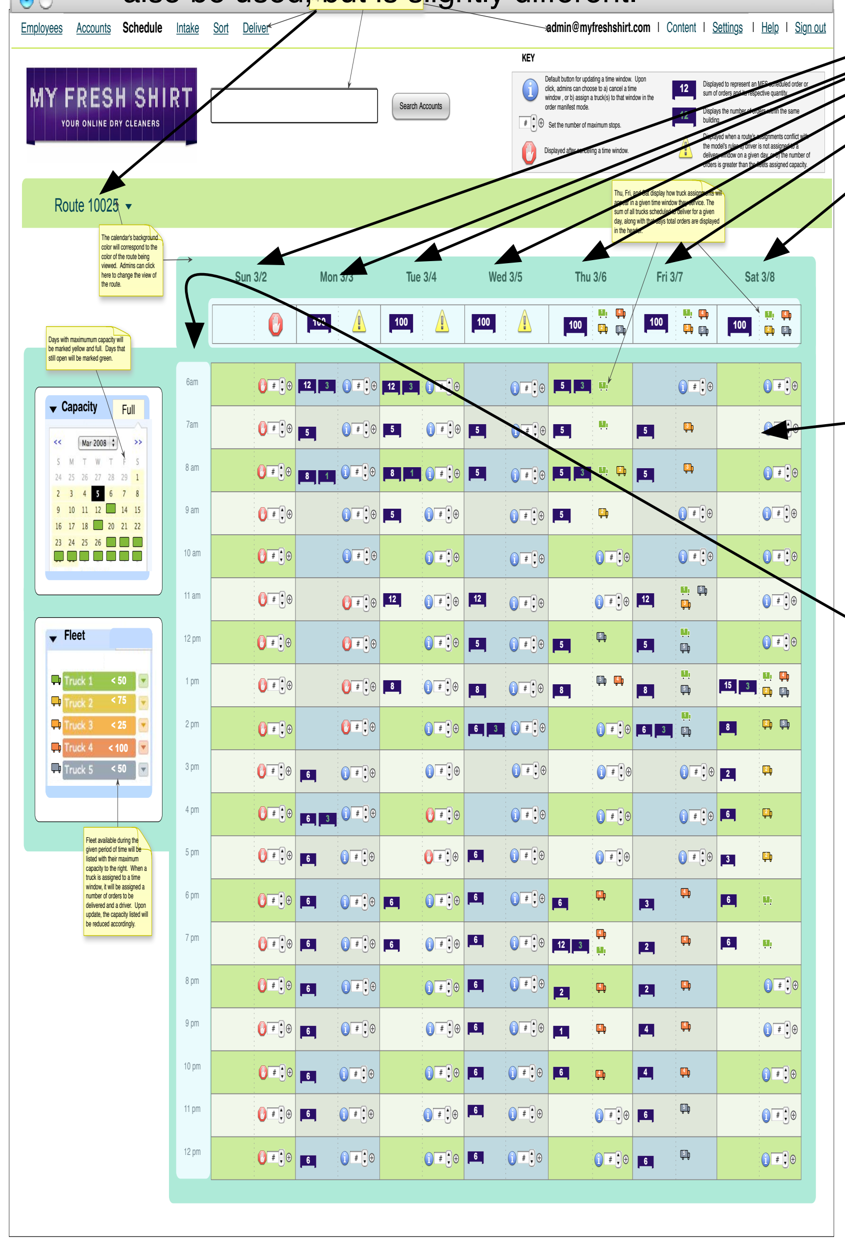
ServicedZip - Each screen corresponds to one and only one of these. This widget is a drop-down selector. Location can also be used, but is slightly different.

Schedule - Each day has its own Schedule obj



mfs_admin_schedule.graffle > Schedule   Wed Mar 05 2008   Modified: Fri Mar 21 2008

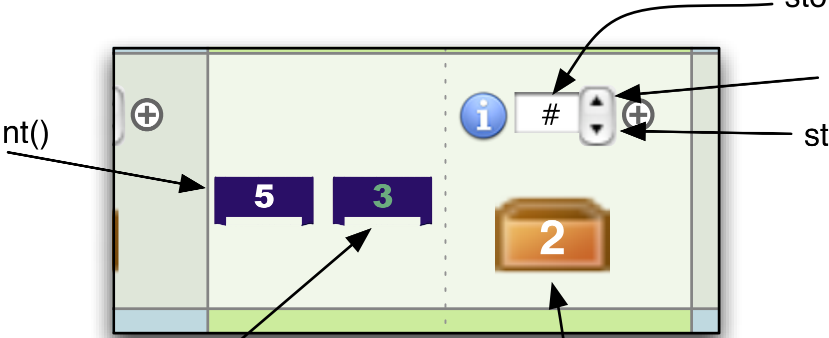
Stop - each of these squares is either a Stop or nil. If there is no service on a route on a particular day at a particular time (and therefore no Stop in the db), it is nil.

Windows - These are the regular time windows as specified in the database. As shown, they are 1 hour long, but for the first version, they will probably be two hours long.

Models of Interest:
Schedule - The scheduling object is an array of stops or nil values
Stops - A stop is an agreement to allow scheduling for a certain route at a certain time
Windows - A window is a period of time (in this case, 2 hour blocks) independent of date
Location - A faux superclass for any location in the system, in this case a ServicedZip
ServicedZip - Corresponds to a route for the purposes of this system.
Truck - represents a truck.

Actions -

1) ask the user what location/servicedzip (route) they want to schedule. This is done via a drop down menu. The list can be accessed via a call to `ServicedZip.find(:all)`.
2) the windows will be constant for now. Use `Window.find_all_regular()`. This will return an array of all the "regular" windows, which is what is being scheduled here.
3) ask the user *when* they are interested in. When the screen first loads, it should default to today. This is controlled by the calendar picker on the left side of the screen.
4) you can find a schedule for a day/route pair by calling `Schedule.for(`*location*`,` *date*`)`. This will return an array of Stops. The array is of the same dimensions as the array of windows you have, so the indexes should line up. That is, if `Window.find_all_regular()[i]` should correspond to `Schedule.for(location,date)[i]`. As a convenience, you can get an array of schedules for a week by calling `Schedule.for_week_of(`*start_date*`)`.
5) if no stops exist, (`Schedule.for(location,date) == nil`), there needs to be some mechanism to create a new one (`Schedule.for(`*args*`)[i] = Stop.new()`) in this area and then save it.
6) The user will be able to edit existing Stops inline. The only attribute the user needs to change here is `stop.slots`. (the lower case 'S' in Stop indicates an instance). The interface will also display the number of customer requests against the stop, as well as its concordance value. These are made available through methods in the Stop instance.



stop.slots

stop.add_slot()

stop.remove_slot()

stop.request_count()

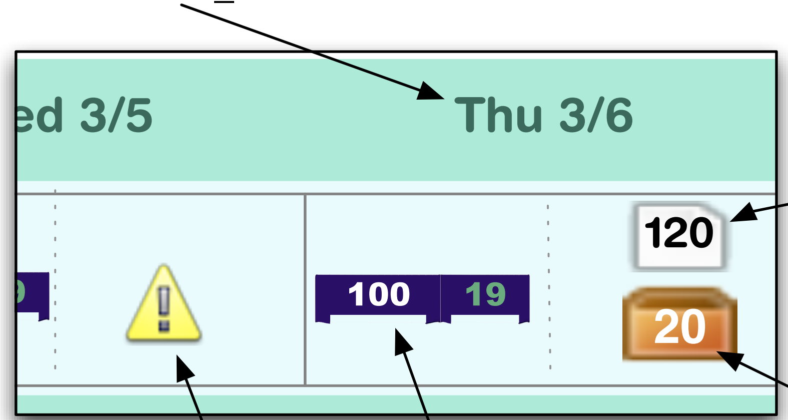
stop.concordant_request_count()

stop.slots_left()

7) So the basic need here is to allow the assignment of slots for each stop in each window of each day. All that's left is the fleet status indicator on the left, which is just a list of available trucks. Call `Truck.find_all_available()`. The color coding is part of the view, so I leave it to you (the model does not indicate colors). The capacity is simply `truck.capacity`.
8) There comes a time when it is no longer feasible to make changes to a schedule. So before you allow any changes to happen, check in with the predicate `schedule.editable?` and don't allow changes to schedules that aren't.

9) Lastly, there are the totals. Though not displayed properly here, you can get the appropriate values through sveral convenience methods of schedule.

`schedule.to_s`



`schedule.total_slots()`

schedule.total_requests()

`schedule.total_concordance()`.

display graphic on error condition