# Conservatory Project Notes

## Exception Handling

rescue\_from – put in application controller to handle exceptions of a certain type (or types) in an entire controller and its subclasses.

e.g. rescue\_from User::NotAuthorized, :with => :user\_not\_authorized

private

def user\_not\_authorized

flash[:error] = “my error”

redirect\_to :back

end

Errors on forms are formatted by default as a div inside a form

<div class=errorExplanation id=errorExplanation>

<h2>2 errors were reported </h2>

<p>There were problems with the following fields:</p>

<ul><li>Admin email taken</li></ul

</div>

## Roles and Privileges

Users have many Roles through Assignments

A Role has many Grants (of Privileges)

A Privilege is Granted to one or more Roles

Users have Privileges through Roles

Privileges are named rights to do something to a resource. They consist of a name, a description, a category, the controller, and the action(s) that are allowed.

On the User model, the “can” method is the way to check whether a user has the right to do something. This is defined as follows:

def can? (action, controller)

unless is\_global\_privilege(action, controller)

roles.includes(:privileges).for(action, controller).any?

end

end

“for” is a named scope on the Role model, which is basically a where clause that filters on the action and controller. We can reference roles for the current user directly as “roles” because of the relationship between User and Role. Is\_global\_privilege references an array of “DEFAULT\_ACTIONS” that are always available to everyone. These include signing in, signing out, accessing the dashboard, accessing help, and editing your personal information.

Scope is always restricted to the current account, unless you are a superuser. An account\_id column that is present on Users, Roles, and Privileges accomplishes this. The superuser defines Default Roles (drawing upon a global set of privileges) that are inherited when a new account is created.

A check\_authorization method is implemented on the Application Controller as a before\_filter, and calls the can? method.

Following is a list of default roles and their associated privileges that are seeded in the system and copied to the account upon its creation.

### Default Roles and Privileges

|  |  |  |  |
| --- | --- | --- | --- |
| **Role** | **Category** | **Privilege (Controller)** | **Actions** |
| System Administrator | Administration | Manage Users (Users) | All |
|  | Administration | Manage Roles (Roles) | All |
|  | Administration | Manage Settings (Settings) | All |
|  | Administration | Manage Email Templates (Messages) | All |
|  | Administration | Manage Locations (Locations) | All |
|  | Registration | Enable/Disable Online Registration |  |
|  | Administration | Manage Account Settings | Billing, canceled, paypal, plan\_paypal, cancel, edit, update |
|  |  |  |  |
|  |  |  |  |
| Registrar | Registration | Manage Time Periods (TimePeriods) |  |
|  | Registration | Manage Registrations (Registrations) |  |
|  | Registration | Manage Group Lessons (BillingItems) |  |
|  | Registration | Manage Private Lessons (BillingItems) |  |
|  | Registration | Manage Schedule Blocks (ScheduleBlocks) |  |
|  | Registration | Manage Time Slots (TimeSlots) |  |
|  | Employees | Approve Hours (HoursWorked, RU) |  |
|  | Payments | Manage Payments (Payments) |  |
|  | Payments | Manage Donations (Donations) |  |
|  | Payments | Manage Credits (Credits) |  |
| Instructor | Students | Manage Private Students (Registrations) |  |
|  | Students | View Rosters (Registrations, R) |  |
|  | Employees | Record Hours (HoursWorked) |  |
|  | Students | View Class Calendar (Schedule, R) |  |
|  | Employees | Request Schedule Blocks (ScheduleBlocks – if owner) |  |
|  | Employees | Manage Time Slots (TimeSlots) |  |
|  |  |  |  |

In the old privileges model, I used this technique to populate a drop down with some values from a hash:

<% controllers\_array = Privilege::CONTROLLERS.map {|priv\_controller| [priv\_controller.first, priv\_controller.last[1]] } %>

<%= f.collection\_select :controller, controllers\_array, :first, :last, prompt: t('privileges.form.selectaresource') %>

## Menus

Menus are generated by querying the MenuItem model and filtering the results based on the privileges (ultimately, the controller/action combinations) for the current user and grouping them by the “category” attribute of MenuItem. This happens as follows: If the session does not contain a :current\_menu entry, the current\_menu(current\_user) method is called on the MenuItem model. This method first checks to see if the user is the owner, if so, then all menu\_items are added to the available list. If not, then for each menu\_item, the can? Method on the user model is called, passing in the controller and action associated with the menu\_item. The “can” method returns true if EITHER the controller/action combination is a “global privilege”, as defined in the GLOBAL\_PRIVILEGES array on the privilege model, OR the user’s roles include privileges for that controller/action combination. The “for” call is defined as a named scope on the Role model, and checks the controller and performs a bitwise comparison of the unique identifying integer defined for that controller/action combo in Privilege::CONTROLLER\_ACTIONS[controller][action] against the actions integer stored with the privilege in the database. When the list of privileges for that user is completed, it is grouped by category, and returned as the “current\_menu” (and placed on the session). When the menu is rendered,

the category is displayed as a large button, and the submenu extends out to the right from the category, displaying the various menu items that are available to the user. The image and text for the category are retrieved from Privilege::CATEGORIES, and the link that is associated with the category button itself is derived from a lookup to Privilege::ROOT\_MENU\_ACTIONS in the helper method link\_for\_privilege(priv\_controller) in application\_helper.rb. Clicking on one of the submenu buttons invokes the action associated with the menu item. Following is a table of menu items. Left Menu Items are described below in the Two Column Layout section.

|  |  |  |
| --- | --- | --- |
| **Category** | **Menu Items** | **Left Menu Items** |
| Administration | People | Import Family(ies) |
|  |  | View Families |
|  |  | Manage System Users |
|  | Account | Modify Organization Name |
|  |  | Change Billing Plan |
|  |  | Edit Payment Information |
|  |  | Edit Account Settings |
|  |  | Cancel Your Subscription |
|  |  | Import Data |
|  | Roles |  |
|  | Settings |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Two Column Layout

The two column layout supports a set of sub menu items beyond the single level of menu items listed by category in the horizontal menu at the top. This is invoked by using the ‘two\_column’ layout, and overriding the left\_menu function that is defined as an empty function (a la an abstract class) in application\_controller.rb. The left\_menu function simply returns a menu\_array, which is an array of two element arrays. Each two element array consists of a string (the menu text) and a link (the link to be redirected to when the menu item is clicked). The left\_menu function is implemented in any controller that needs one. Help text can also be displayed on a per controller basis in the form of a post-it that will appear underneath the left menu. This is implemented in a similar way. The controller declares a private function called “help\_text”, which overrides an empty declaration in application\_controller.rb. In the left menu view, a helper function (in application\_helper.rb) called “helpful\_information(help\_text)” is called unless help\_text is nil. So, if help\_text is set in the current controller, this helper function will display the appropriate help text for that controller. It will do so UNLESS the help\_text is nil (not defined in the current controller) OR a user\_preference has previously been set for that particular help. The user preference is set when a user sees the help, then clicks on the X button in the upper right corner of the help. This is done via an ajax call, and the help is then immediately hidden. **Open question**: Provide a screen where a user can manage all of his/her help post-its? That would be the only way for a user to see the help again after hiding it the first time.

## Models/Controllers

Semester/Session

System Preference

Family (customer)

User

Role

Privilege

Location

Registration

Event

Billing Item

Schedule Block

Time Slot

Invoice (and Invoice Line Item)

Sales Receipt (and Sales Receipt Line Item)

Payment

Credit Memo

Time Card

## Styles

Global styles are defined in the application style sheet, with styles that are specific to controllers defined in controller specific style sheets. There are global styles defined for Lists, Forms, and Buttons. Buttons have gradients that are defined using the –webkit-gradient style, which takes a From color and a To color. There are three states of the buttons: Normal, Hover, and Active which is the style shown when the button is clicked. Normal and Active just reverse the colors and change the box shadow slightly.

## Workflow for setting up a new account

1. A new customer browses the public site and selects free trial (or a paid subscription)
2. The new customer fills in information about the account and about the administrative user (the Account owner) for the account.
3. Account owners are exempted from the authorization checking, and are therefore “account superusers”. It is possible to grant other user semi-superuser privileges (except ownership changing) through roles (e.g. the System Administrator default role or a custom role).
4. When the customer saves the information, a new account is created, a new user is added to that account, and a default set of administrative privileges are granted to that user. Also, a default set of Roles are assigned to the account so that they can be granted to other users that the administrative user may create.
5. The account owner for the new account signs in, and can begin adding new users and granting them Roles. The Administrative user can also define new Roles that are specific to her account if she likes (this should probably be pretty unusual).
6. The account owner also sets account settings to personalize their account, e.g. the term used to describe a semester or a student, and the email address(es) to be used for alerts.
7. The account owner can optionally reassign “ownership” of the account to another person. When the change is saved, only the new owner (or the superuser) can then reverse the change.

## Steps to add a new listing page:

1. Controller: Add respond\_to :js, :only => :index and helper\_method :sort\_column
2. Controller: In protected section, add:

def collection

@<collection name> ||= end\_of\_association\_chain.search(params[:search]).order(sort\_column + ' ' + sort\_direction).paginate(:per\_page => 15, :page => params[:page])

end

1. Controller: in private section add:

private

def sort\_column

<model name>.column\_names.include?(params[:sort]) ? params[:sort] : "1"

end

1. Model: add the following and adjust the search:

def self.search(search)

if search #&& column\_name && self.column\_names.include?(column\_name)

where('name LIKE ?', "%#{search}%")

else

scoped

end

end

1. Add index.js.erb with the ajax call: $('#roles').html("<%=j render 'role\_list' %>"); (substitute the right ID’s)
2. Add a partial \_role\_list.html.erb to display the table and facilitate ajax
3. Add the shared menubar partial to the index.html.erb file

## Subscriptions

Billing options will include yearly at $1,500.00 per year, or monthly at $150.00 per month. The customer will be able to sign up for a free 30-day trial prior to being billed. When they sign up, they will not be asked for a credit card, but will be notified to convert 2 weeks in, 1 week out from the end, and daily for the last 3 days. To sign up for the trial, they will need to provide the name and address of their organization, as well as the name and email address of the account owner. If the customer signs up for a paid plan, their credit card or paypal account will be charged immediately.

## User Limits

The account model calls has\_subscription, which looks for \*\_limit fields in the subscription table (and the subscription\_plan table, which populates the value in the subscriptions table upon account creation). If there is a value in the limit field, this value is checked by this function in the User controller and in the New User view. If the number of users is updated in the plan after a subscription is created, that subscription is not affected.

## Miscellaneous

Made a change to rails3\_acts\_as\_paranoid.rb 0.1.3 to eliminate DEPRECATION warnings. #alias\_method :foreign\_key, :primary\_key\_name unless respond\_to?(:foreign\_key)

alias\_method :foreign\_key, :primary\_key\_name unless instance\_methods.include?(:foreign\_key)

warnings

Cool technique for implementing search on attributes:

class Post < ActiveRecord::Base

class << self

# Search the title and body fields for the given string.

# Start with an empty scope and build on it for each attr.

# (Allows for easy extraction of searchable fields definition

# in the future)

def search(q)

[:title, :body].inject(scoped) do |combined\_scope, attr|

combined\_scope.where("posts.#{attr} LIKE ?", "%#{q}%")

end

end

end

end

Getting Has One nested form to work (e.g. Primary Contact for a family) required setting the following in the Family model:

1. has\_one :primary\_contact, :class\_name => "User", :conditions => { :is\_primary\_contact => true }
2. accepts\_nested\_attributes\_for :primary\_contact
3. attr\_accessible :name, :is\_individual, :primary\_contact\_attributes (note the \_attributes suffice and the fact that I have to make all the family attributes accessible)
4. Also, to make fields\_for work, I had to use a before\_filter in the families\_controller to build the User (primary contact)