IST488/688 Social Web Technologies, Fall 2011
Lab 8.
Instructor: Keisuke Inoue

Description:

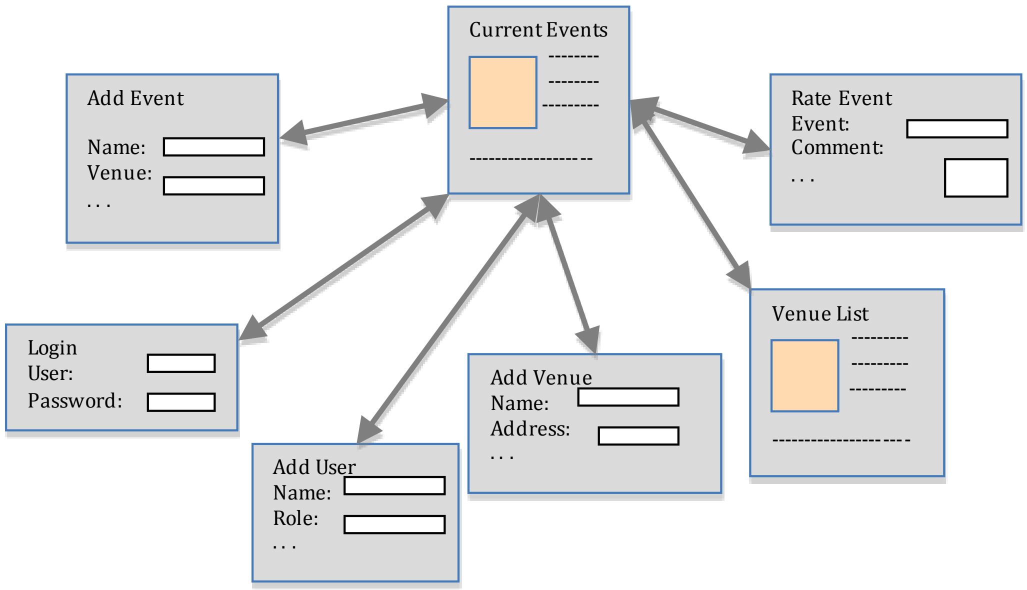
In this lab, you will continue to work on Ruby on Rails, a web application framework based on Ruby. **Your work in this lab will be a continuation of Lab 7. So make sure you have finished Lab 7, before you proceed.** The information you will need is this lab is covered in Ruby on Rails book, Chapter 7 (Validation) Chapter 14 (User Model). If you have a question, feel free to ask the instructor or the assistant.

Note: this lab assignment is designed as an individual work. To maximize your learning experience, you are encouraged to work on it independently. You may ask questions to instructors and/or friends (but keep it quiet) and look up resources, but you are not allowed to copy-and-paste any resources other than you created.

Scenario (recap):

As part of the Connective Corridor initiative, you are building a social web system that promotes cultural and artistic events (e.g. art exhibitions, theatrical plays, etc.) in the area. The system will provide the platform for the local community residences and students to share or recommend such events or venues, through the web and mobile devices.

A simplified version of the page flow will be something like this (The graphics is just to remind you of the description that was provided in the previous class):



We have been working on the content, events and venue data. Now we are moving onto user management.

Lab Instructions:

1. Basic Setup

1.1. First, login to the Linux system through itellv.ischool.syr.edu, launch a terminal application go to the app directory that you worked on last week (i.e. type “cd apps/cc”).

1.2. Now, before starting to working on let us start using git. Git is a simple version control system that lets you manage the different versions of your source code. For our purpose, it will be useful, if you break your app really bad and want to go back to the stage where everything was working. You can read more about the version control system at http://git-scm.com/. For now, type in the following command:

```
git init

git add .

git commit –m "Upto Lab7"
```

The first command will initialize a repository; the second command will add all the files in and under the current directory to the list of files that need to be kept track of; and the third command will "commit" (store the current files that have been changed since the last "commit" or "init" in the repository).

1.3. If the Rails server is not running, run the server in the cc directory.

2. Adding User Model

So far, our system is open to public for all the CRUD operations. (What is 'CRUD'…? You must know it if you have taken the database course! It is an acronym for the most basic database operations: Create, Read, Update, and Delete… in Rails' actions, they are called "new", "show" (or "index"), "edit" and "destroy" respectively.) Anyone can delete all the records in the database from the web. We will add user management to fix this situation.

2.1. Where shall we start? The first thing we need is a model for the users. So let's start from scaffolding. Generate a scaffold for the User model, with the following attributes:

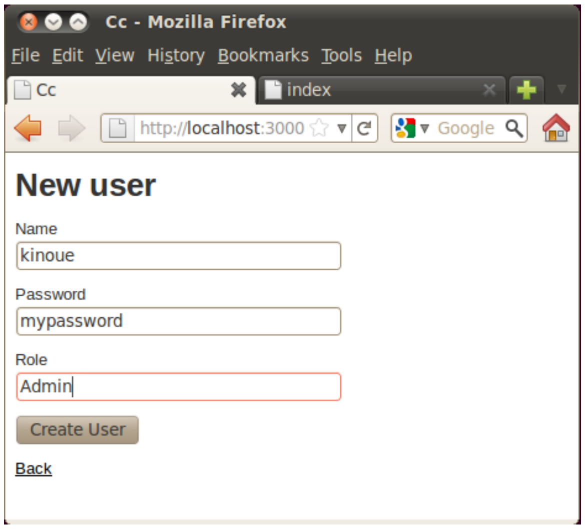
- name: String
- password: String
- role: String.

(You must know which command you have to execute here.)

Once it's done, update the database using rake as usual. (Again, you must know which command you have to execute here.)

In real world systems, you never store passwords in database. Instead, you store hash values of passwords, for (obviously) security reasons. In this lab, we are going with a less secure approach for simplicity. Chapter 14 of the Ruby on Rails book explains the alternative (better) approach using a hash function. You are welcome to use the alternative technique in the final project, but let's keep it simple for now.

2.2. Now, launch Firefox and go to http://localhost:3000/users/new. Let's see what happens if we create a new user…



New user

Name
kinoue

Password
mypassword

Role
Admin

2.3. Crap! It shows the user's password right there!! We must fix that… While we are doing it, let's do some other modifications too. Open the user's form file (You must know which file.), and modify it as follows (the bold font indicates modifications):

```
<% form_for(@user) do |f| %>
  <% if @user.errors.any? %>
    … Error handling here …
  <% end %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </div>
  <div class="field">
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </div>
  <div class="field">
    <%= f.label :role %><br />
    <%= f.select :role, ['Admin', 'User'], :prompte => 'Select Role…' %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
<%= link_to 'Back', users_path %>
```

Notice that we used password_field, instead of text_field, so that the form won't show the typed password on the screen. We also made two additional modifications:

1) Added the password_conformation field, to make sure the password was correct.
2) Modified the role field to a dropdown menu, in order to limit the possible values of the field. (This should actually be done using a separate database table, say `roles`, but we are going with a quick and dirty approach here again.)

2.4. This form is assuming that there is a new attribute in the User class, named password_confirmation, so let's implement it. Notice that this is a new trick… the password_confirmation attribute **should not be in the database** (because what is the point of storing two same passwords, right?), so it was not and should not be part of the scaffolding. From the point of view the database, the User class has only three fields: name, password, and role… the confirmation is just for the sake of confirmation, and no more. So how should we include the attribute in the class definition…? We could use our old friend, attr_accessor, but that will make password_confirmation visible to outside the class. So let's use the public and private declarations that we learned in the previous class. Open the User model file and include the following line in the class definition:

```
private
def password_confirmation
  @password_confirmation
end

public
def password_confirmation=(pw)
  @password_confirmation = pw
end
```

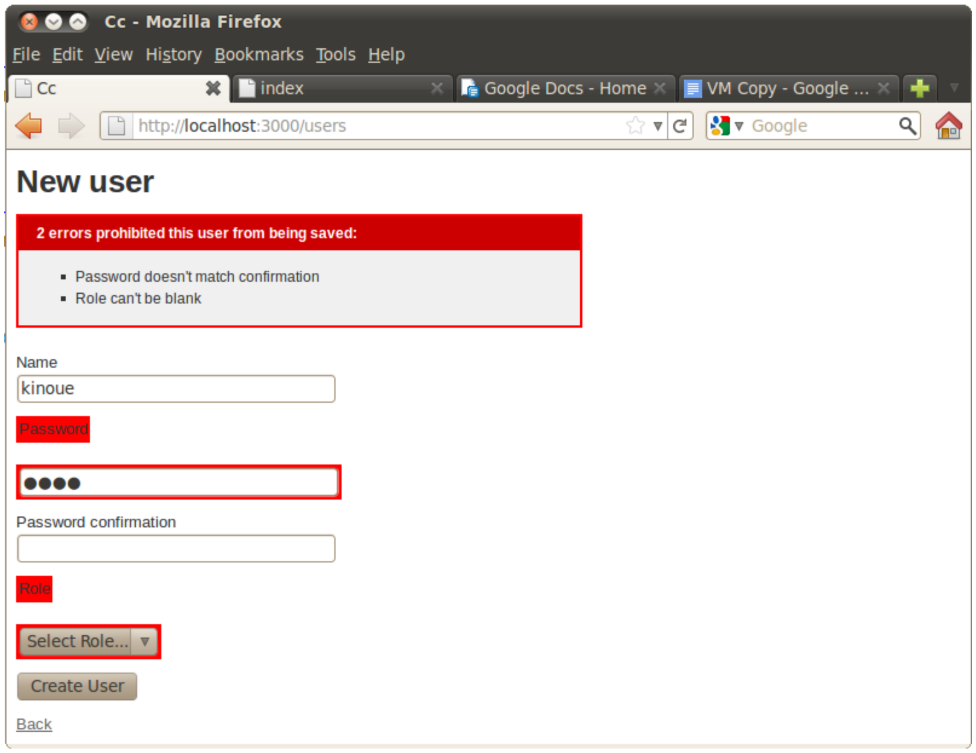
These separate definitions of the writer and reader methods allow other classes to set the value of the password_confiramation attribute, but not to read it. Perfect!

2.5. While you are at it, let's add some neat features that Ruby on Rails provides. Modify the User class definition further, so that it will looks like below:

```
class User < ActiveRecord::Base

   private
   def password_confirmation
     @password_confirmation
   end

   public
   def password_confirmation=(pw)
     @password_confirmation = pw
   end

   validates :name, :presence => true, :uniqueness => true
   validates :password, :presence => true, :confirmation => true
   validates :role, :presence => true

end
```

Each line that starts with "validates" tell Ruby on Rails that it needs to validate the specified nature of the field before it processes it further. For example, if the `:presence` parameter is set to true, it means the field must not be blank. If the `:uniqueness` parameter is true, the value of the field must be unique in the database. Furthermore, if the `:confirmation` parameter is true, Ruby on Rails will conveniently compare the given field and its confirmation field (of which name Rails guesses correctly!) and make sure they are the same.

2.6. Once you save the file, go to Firefox and try creating a new user, but with some empty fields or passwords mismatched. Rails should give you error messages as shown below:



New user

2 errors prohibited this user from being saved:

- Password doesn't match confirmation
- Role can't be blank

Name
kinoue

Password
●●●●

Password confirmation

Role
Select Role...

Create User

Back

2.7. **Once you know that validations are working, create at least one user with the "Admin" role, so that you can login with the user, before moving on to the next step.**

3. Implementing Authentication Process

Now that we have the User database setup, we can add an authentication process to our web system.

3.1. First, we need actions, say, login and logout. Let's define them in a controller, called admin. Generate the controller file with the following command:

```
script/rails generate controller admin login logout
```

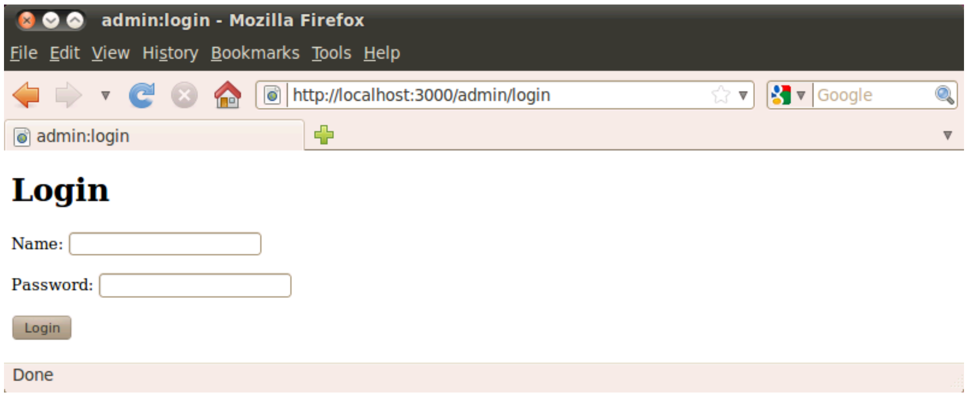
This command will generate template files for the admin controller (app/controller/admin_controller.rb), as well as the login viewer (app/views/login.html.erb) and the logout viewer (app/views/logout.html.erb) (and a few more files that we don't use in this lab).

3.2. Edit the login viewer file (app/views/login.html.erb) as the following. We are using a few unique tags that are provided by Rails, as opposed to the standard HTML tag, but they are self-explanatory -- form_tag genates HTML form tag, text_field_tag generates an HTML input tag with type="text", etc. Notice that each text/password input field specifies the hash variable and key. This hash is used to communicate with the login action (You will see how it is done in the next step).

The submit_tag will be an HTML input tag (type="submit"), which jumps to login.html, the same page, but with the HTTP POST request.

```
<h1>Login</h1>
<% form_tag do %>
    <p>
      Name: <%= text_field_tag :name, params[:name] %>
    </p>
    <p>
      Password: <%= password_field_tag :password, params[:password] %>
    </p>
    <p>
      <%= submit_tag "Login" %>
    </p>
<% end %>
```

On Firefox, the page should look something like below (you can make it fancier, if you would like):


admin:login - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:3000/admin/login
Google
admin:login
Login
Name:
Password:
Login
Done

If you have the Rails book with you, check out Figure 14.2 on page 194. This explains how login.html.erb and admin_controller.rb interact. Don't try to log in now, though. It won't work, yet…

3.3. Now that we have the viewer for the login action, we need to implement the corresponding method. Open the admin controller file and edit the login action as follows:

```
def login
  if request.post?
    user = User.find_by_name(params[:name])
    if ! user.nil? && user.password == params[:password]
      session[:user_id] = user.id
      redirect_to(:controller => 'events', :action => ‘index’)
    else
      redirect_to(:back, :alert => “Invalid user name/password.”)
    end
  end
end
```

The code may looks scary at first, but it is not as bad as you might think…

The first expression is an if statement that checks if the method is called from a POST request. This checks if the method is called because the login.html was accessed (GET request) or the form was submitted (POST request). In the former case, the method has nothing to do, because the page should be displayed simply. In the latter case, the method needs to perform the authentication process.

Now assuming that the request was indeed a POST request, the method attempts to find the user model from the database, given the user’s name, provided from the form. The class method, find_by_name, is a method that is automatically created by Rails, given that “name” is an attribute of the User class. Notice that we are using the `params` hash to access the input from the user.

The next expression is another if statement that checks if the variable `user` has a non-nil value, which means the user of that user name was found, and the password that was typed in matches with the one in the database. If the conditions are met, the method stores the user ID in an HTTP session variable, again, using a hash key. (HTTP session is a mechanism (application layer protocol) that associates web servers and individual visitors. Sessions can be implemented in different ways, e.g. using cookies (default) or using a database on the server. For more information, see Chapter 20, page 323 of the Rails book.)

The `redirect_to` function namely redirects the user to another action, in this case, /events/index.html, but it could be anywhere else.

Lastly, in case the conditions are not met, it will go back to the current pay, displaying an error message using a convenient parameter, `:alert`.

3.4. Once you have saved the controller file, open a routing configuration file (config/route.rb). In the top of the page you should see something like the following:

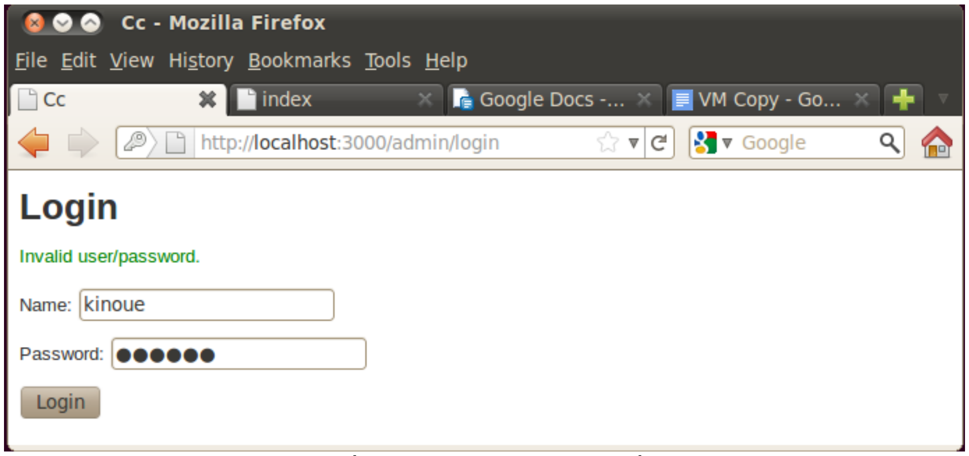
```
CC:Application.routes.draw do
  get “admin/login”
  get “admin/logout”

  …
```

The exact mechanics of the routing will be discussed in the later class. For now, what we can tell is that the code seems to be handling get requests for the login and logout methods. However, as explained above, you also need to handle post requests to the login method. So let’s add a simple line saying:

```
post “admin/login”
```

Save the file.

3.5. Now, go to FireFox and open the login page, i.e. http://localhost:3000/admin/login. Go ahead and try logging in. (You may want to test what happens if you fail to log in as well.)


Cc - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Cc
index
Google Docs -...
VM Copy - Go...
http://localhost:3000/admin/login
Google

Login

Invalid user/password.

Name: kinoue

Password: ●●●●●●

Login

Testing a wrong password…

If you succeed, you should see the event listing page.

4. Logout Action

Since we implemented the login action, let's implement the logout action. This is really simple -- since we don't need to take any inputs from the user, we don't need to implement the viewer. (You can, if you would like to.) What we need to do is to define the logout method in the AdminController class (app/controllers/admin_controller.rb), so that the method will erase the user_id variable stored in the session. The following code will do the job:

```
def logout
  session[:user_id] = nil
  redirect_to :controller => 'admin', :action => 'login'
end
```

5. Limiting Access

The login mechanisms are meaningless, unless they are enforced. Here, Rails offers you yet another convenient way to implement it, called **filters**. Filters are methods that called before and/or after actions are invoked. In our case, we need to set up a **before_fileter** for all the actions in our site, so that visitors will go through the authentication process before entering a certain pages. (I said "all the actions", but more precisely, all the actions except the login action... otherwise visitors will end up in an infinite loop, right?)

5.1. On the text editor, open the application controller file (app/controllers/application_controller.rb). First, add the following line in the definition of the ApplicationController class.

```
before_filter :authenticate,  :except => 'login'
```

This line specifies that Rails will run the authenticate method before all the actions, except the login action.

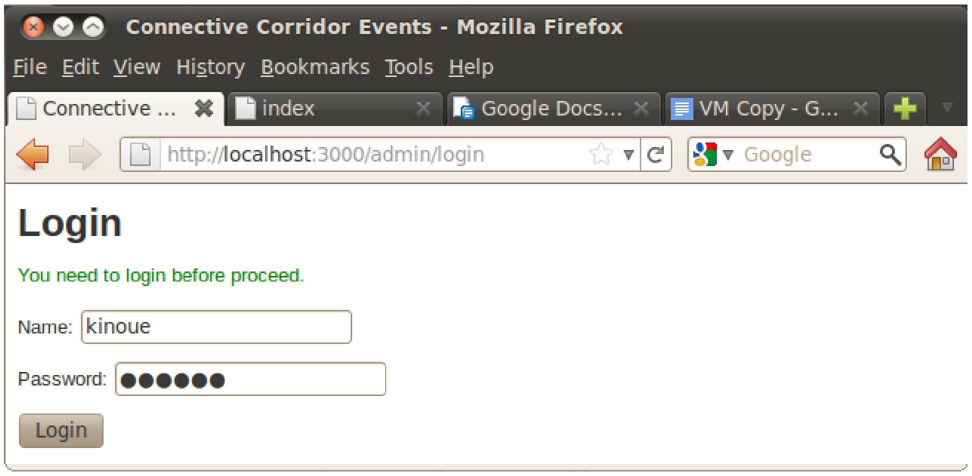
5.2. What should the authenticate method do? The authenticate method should check if a user has been logged in, and if he/she hasn't, force him/her to go to the login page. Here is a definition:

```
def authenticate
  if session[:user_id].nil?
    flash[:alert] = 'You need to login, before proceed.'
    redirect_to :controller => 'admin', :action => 'login'
```

```
      end
    end
```

As you defined in the login method, if a user successfully logs-in the user's ID is stored in the session variable… so that is what you can use to check if the user is logged in or not.

flash is another special hash variable used in Ruby on Rails. It is used to display a message that needs to be displayed only once. (In other words, the message will be deleted once it's displayed.) You need to define this method in the ApplicationController class. Once it's done, logout from the system, by accessing to http://localhost:3000/admin/logout, then access to one of the actions, e.g. http://localhost:3000/venues. You should be automatically forwarded to the login page.


Connective Corridor Events - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Connective ...
index
Google Docs...
VM Copy - G...
http://localhost:3000/admin/login
Google

Login

You need to login before proceed.

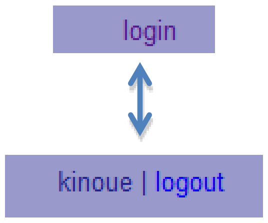
Name: kinoue

Password: ●●●●●●

Login

6. Displaying the user name using Layout

So now the authentication mechanism is somewhat functional, but there is no way to see if the user is currently logged-in or not. In many web services there is usually an element displayed in the page, that displays a hyperlink to the login page, if the user is not logged-in, and the user's name and a hyperlink to the logout action, if the user is logged-in. Let's implement that before we further improve the authentication mechanism.



For this functionality, you will edit not an individual viewer file (i.e. xxx.html.erb), but the layout file. Open the file views/layout/application.html.erb. It should look as below:

```
<html>
 <head>
  <title>Cc</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
 </head>
 <body>
  <%= yield %>
 </body>
</html>
```

This is the template of all the pages rendered by Ruby on Rails -- the line `<%= yield %>` means that this portion is replaced by rendering from individual viewer files, depending on the routing. Any code you add here will be effective to all the pages you see. For a test, change the title field to “Connective Corridor Events”. You should see the title of the all pages will become “Connective Corridor Events”.

Now, here comes the difficult part. Add a segment of code, in the body of the file, that does the following:

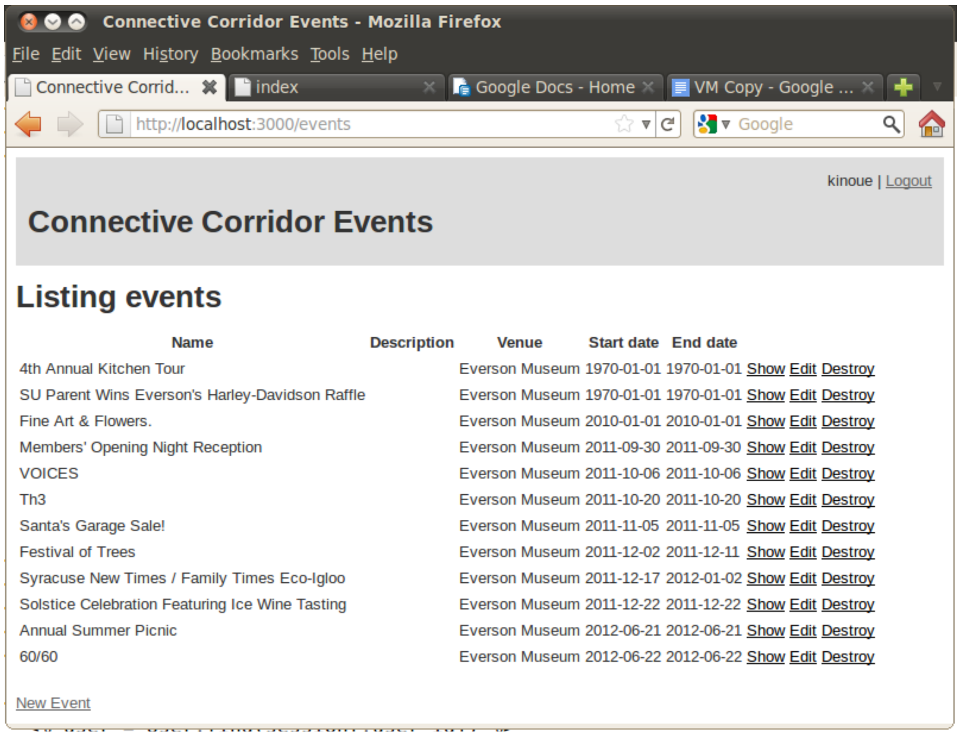
1. Check if the user has been logged in.
2. If yes, display the name of the user, and the hyperlink to the logout action.
3. If no, display the hyperlink to the login action.

I am hoping you can figure out how to do these, but if not, here are some hints:

1. Check out the authenticate method that you created above. See how you can check if the user is logged in.
2. If the user is logged in, you know the user’s ID, right? So how can you know the name of the user? Remember, you can always retrieve the user object from the database using the `find` method.
3. In order to display a hyperlink to a certain action, you can use the `link_to` method. For example, the following code will generate a hyperlink to the logout action.

```
<%= link_to ‘Logout’, :controller => ‘admin’, :action => ‘logout’ %>
```

If you know CSS/HTML, you can define some styles and format the page. Once it’s done, the page should look something like the below (using CSS/HTML is totally optional).


Connective Corridor Events - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Connective Corrid...
index
Google Docs - Home
VM Copy - Google ...
http://localhost:3000/events
Google

kinoue | Logout

Connective Corridor Events

Listing events

Name	Description	Venue	Start date	End date	
4th Annual Kitchen Tour		Everson Museum	1970-01-01	1970-01-01	Show Edit Destroy
SU Parent Wins Everson's Harley-Davidson Raffle		Everson Museum	1970-01-01	1970-01-01	Show Edit Destroy
Fine Art & Flowers.		Everson Museum	2010-01-01	2010-01-01	Show Edit Destroy
Members' Opening Night Reception		Everson Museum	2011-09-30	2011-09-30	Show Edit Destroy
VOICES		Everson Museum	2011-10-06	2011-10-06	Show Edit Destroy
Th3		Everson Museum	2011-10-20	2011-10-20	Show Edit Destroy
Santa's Garage Sale!		Everson Museum	2011-11-05	2011-11-05	Show Edit Destroy
Festival of Trees		Everson Museum	2011-12-02	2011-12-11	Show Edit Destroy
Syracuse New Times / Family Times Eco-Igloo		Everson Museum	2011-12-17	2012-01-02	Show Edit Destroy
Solstice Celebration Featuring Ice Wine Tasting		Everson Museum	2011-12-22	2011-12-22	Show Edit Destroy
Annual Summer Picnic		Everson Museum	2012-06-21	2012-06-21	Show Edit Destroy
60/60		Everson Museum	2012-06-22	2012-06-22	Show Edit Destroy

New Event

7. Improving the Authentication Mechanism

Now that we can see the user’s loggin-in status, let’s improve the authentication mechanism. Open the application controller file again, on your text editor.

7.1. First, currently, the users cannot view any pages unless they login, which is pretty inconvenient. So let’s modify the line we inserted earlier as follows:

```
before_filter :authenticate,  :except => [‘login’, ‘index’, ‘show’]
```

This will make users to be able to access other actions (across the controllers) without logging in.

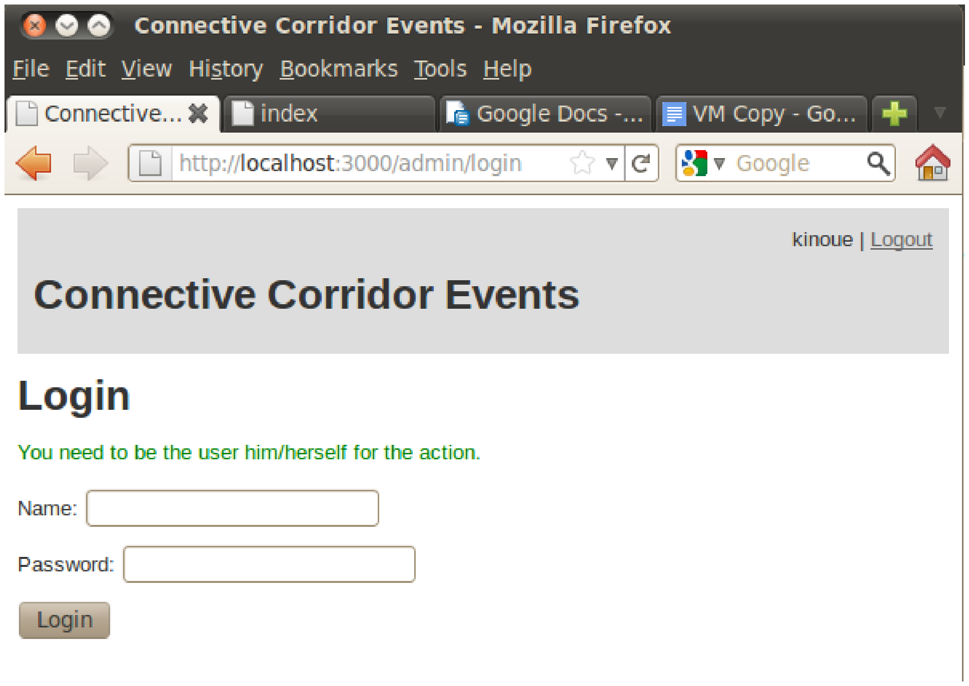
7.2. Second, currently any user can modify any user's information, which is not good, obviously. So Let's make a change so that only the user him/herself can modify the his/her information. In order to do that, you need to add another filter method -- let's call it authenticate_self -- to the update action of the user controller. Add the following lines to the definition of the UsersController class:

```
before_filter :authenticate_self,  :only => 'edit'

def authenticate_self
  if session[:user_id].nil?

    flash[:alert] = 'You need to login before proceed.'
    redirect_to(:controller => 'admin', :action => 'login')

  elsif session[:user_id].to_s != params[:id]

    flash[:alert] = 'You need to be the user him/herself for the action.'
    redirect_to(:controller => 'admin', :action => 'login')

  end
end
```

All the lines should make sense – this time, instead of the `:except` parameter, the `:only` parameter is used. And since this is defined in the `UsersController` file, the filter applies only to the edit action of the `UsersController`. The `aunthenticate_self` method is very similar to the `authenticate` method we already defined, except it also checks if the current user id matches to the user id of the user that is about to be edited.

Notice the use of `param[:id]`, in order to grab the user id of the user that is about to be edited. Also, notice the `to_s` method for the `session[:user_id]`. This is a bit awkward trick that is needed because the types of the two values were different.

Once it's done, create another user and test if it works.


Connective Corridor Events - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Connective...
index
Google Docs -...
VM Copy - Go...
http://localhost:3000/admin/login
Google
kinoue | Logout
Connective Corridor Events

Login

You need to be the user him/herself for the action.

Name:

Password:

8. Bonus Question (5 points)

Currently, we are not utilizing the roles of users. Modify the code for the Step 7, so that users with the Admin role can modify other users' information, in addition to the user him/herself. You can rename the authentication method `authenticate_self_or_admin`.

9. Submission

1. On your terminal window, type in the following commands (this will only update your repository):
   ```
   git add .
   git commit –m 'Lab 8'
   ```
2. Then, go to the rails_apps directory (e.g. type in 'cd ..').
3. Type in the following command:
   ```
   tar cvfz lab8.tgz cc
   ```
4. Now, launch Firefox on your virtual machine.
5. Go to: http://sanka.syr.edu/swt/ on Firefox, login, and submit the file that was created in the previous step.