How to Create the Inventory App on Rails 3.1.1

Last edit: 11/23/2011

About the Course

Accelerated Rails is a 6-week course covering Ruby on Rails. We meet online, once a week, for 90 minutes. The course is intended for profesional or semi-professional developers wanting to get up to speed with Ruby on Rails (RoR). You should be proficient in at least one other programming language, comfortable with object oriented programming, and familiar with internet technology fundamentals including HTTP, client-server relationships, and html/css. We assume you can pick up the Ruby language quickly because, while we will certainly use the language, we won't be teaching it.

There are a number of books you could consult as well as a wealth of online materials. Be sure the materials are for Rails 3. A lot depends on your taste and discipline. Here are some recommendations.

"Ruby on Rails 3 Tutorial", Michael Hartl, Addison-Wesley
"The Rails 3 Way", by Obie Fernandez, Addison-Wesley
"Agile Web Development with Rails", Fourth Edition, by Ruby, Thomas, and Hansson, Pragmatic Bookshelf.
"Rails Antipatterns", Pytel and Saleh

Ryan Bates' RailsCasts (online and podcast)
Rails Forum: http://railsforum.com/index.php

Your software environment should include:

Ruby 1.9.2
Rails 3.1.*
Google+ account for live streaming of the course
A database client such as SQLite Administrator or similar for MySQL is handy.

About the application we will develop

The course is structured as a walkthough to making a Rails 3 application. Your own app needs are probably differenet from the one we will develop, but if you have never developed a Rails app this will show you a lot of the basics that most any app will need.

As the code is developed it will be pushed to github, under the name **jamesgaston**.

The app was inspired by the fact that I've moved a few times in the past two years and have had to decide what to move, what to sell, and what to give away. This app is envisioned as the foundation of a sort of social media app for "stuff". I know apps like this exist (I have one on my android), but coming up with something revolutionary is beyond the scope of this course :-)

The strategy of attacking this problem is as follows:

1. Plan the application
2. Create the app
3. Create static pages

4. Edit the layout file
5. Add support for users (new user, user login, user logout, edit user profile)
6. Add support for Sessions (login, logout, persistent connection)
7. Cancelling a form
8. Add business logic
    a. Add support for our inventory items: new, edit, delete, index
    b. Add lookup tables for inventory item categories and ownership status
9. Add lookup tables for inventory items (category, ownership status)
10. Add lookup tables to for User model (province-state)
11. Replace link_to with button_to for delete item
12. Add some css
13. Add support for paid and admin users
14. Change admin implementation
15. Add ability to edit lookups and add new ones.
16.
17. Email registration confirmation (email invitations?)
18. Add support for controlling the provinces lookup table
19. Add social support: allow users to search for users to sell/trade/aquire stuff
20. Enhance the Items table: perhaps we need size, condition, value, and/or other fields? Or links to item-specific fields?
21. Improve the UI with images, jQuery, Ajax
22. Testing

About this document.

This document chronicles the creation of a Rails 3 webapp. In the course of developing the app, its behaviour and implementation will evolve. In other words, sometimes I write come code to get something working, then go back and revise it to make it work better.

1. Planning the Application

Many apps need pages such as about, help, signin, signout, contact, etc. Our app will not be storing anything in a database for these pages, so these will be **static pages**.

Pages: serves up static pages for contact, help, home (greeting if not signed in), and about.
Model:
Controller: about, contact, help, home
Helper:
Views: about, contact, help, hom

Most apps need to support **users**, and need to store, at a minimum, a user's email address and encrypted password for identification purposes. We will add a few additional fields for users -- such as name, phone, and location -- but if your users vary in type of information associated with each you could have separate models for different types of user and a foreign key into the users table. For the inventory app, all users are the same (aside from the admin) so everything will go in the user model.

User: methods to allow user to edit their information. Admin can see all users, edit all user data, upgrade the user type, and delete a user.

Model: all info about user
Controller: new,create,edit,update,delete, index [admin], edit and delete
Helper:
Views: new, edit, index

If you allow users to login and logout, you need support for **sessions** so we will create a sessions controller with new, create, and destroy methods

Finally, we need to identfy the data needs for our business logic. The inventory app will start with an **items** table plus several lookup tables.

Lets break the app up into its resources, which translates to architecural components and behaviour.

Session: login, logout, saves session-related information, verify logged in.

Model: none, there is no permanent info store about sessions
Controller: new, create, destroy
Helper: sign_in, current_user=(user),current_user, sign_out, signed_in?
Views:

Item: data + view, edit, change [admin access for “all” items].

Model:
Controller:
Helper:
Views:

Pages:

2. Create the App

Generate a new rails application.

rails new <appname> [--database=mysql | oracle | sqlite3 | etc.]

By default, the app will be in the <appname> directory. Sqlite is the default database, otherwise one must be specified on the command line. See config/database.yml which defines a database configuration for each of the three tyes of environments: development, test, and production.

If using Rails 3.1.1 you may need to add a couple of gems.

rails -v   # shows rails version number

```
gem install execjs
gem install therubyracer
```

Now edit Gemfile (in app's root directory) and **add the line**

```
gem 'therubyracer'
```

Next we'll install the gems that are specified by Gemfile, then write a gem snapshot to Gemfile.lock. The defaults are fine for now but you might want to take a look at both Gemfile and Gemfile.lock (in app's root directory)

```
bundle install
```

Run and test the application.

```
cd <appname>
rails server
```

View the app's /public/index.html file by opening a browser and poing to http://0.0.0.0:3000

3. Create the Static Pages

Since our static pages are not storing anything in the database all that is needed is a controller, a view for each page, and routing information in config/routes.rb. Therefore all we need is a controller, some view templates, and some routes.

These are the files we need to create:

app/controllers/**pages_controller.rb**
app/views/pages/**about.html.erb**
app/views/pages/**contact.html.erb**
app/views/pages/**help.html.erb**
app/views/pages/**home.html.erb**

To generate the controller for our static pages use this command.

```
rails generate controller Pages home about contact help
```

Note that the controller name Pages is **plural.**

When you run this command you'll see that additional files are created (tests, helpers, javascript and stylesheet assets) but we will get to those later. Before you start invoking the Pages controller's methods you should think about the entries in the routes file. For now, take a look at config/routes.rb and you'll see that the controller added an entry for each page, in the form:

```
get "pages/home"
get "pages/about"
get "pages/contact"
get "pages/help"
```

For our app I've decided to use the following syntax in config/routes.rb for the static pages:

```
match '/contact', :to => 'pages#contact'
match '/about', :to => 'pages#about'
match '/help', :to => 'pages#help'
root :to => 'pages#home'
```

Take a look at the Pages controller (app/controllers/pages_controller.rb). We'll leave it as is since all it needs to do is pass control from a controller method to the matching view. Do note that there is a controller method for each view we specified on the generate controller command line. You might also note that sometimes we don't need all the views generated because we may redirect in the controller our routes file to some other action.

Now you can edit the home page (app/views/pages/home.html.erb) so it says something of interest.

One more thing: Remove the file **public/index.html.** We'll let the router control what is loaded.

4. Edit the Layout File

The controller is responsible for handling a request, and when it is time to send a response back to the user it hads control off to the view. If this response is a full-blown view, Rails wraps the view in a layout, and may even pull in partial views. Layouts and views are just erb files, like the view itself. Alternatively, Rails could be sending back an HTTP header, JSON, XML, etc., but here we'll focus on the scenario where Rails is sending back a full page view.

To find the current layout, Rails first looks for a file in app/views/layouts with the same base name as the controller. For example, rendering actions from the PagesController class will use app/views/layouts/pages.html.erb. If there is no such controller-specific layout, Rails will use app/views/layouts/application.html.erb. Rails also provides several ways to more precisely assign specific layouts to individual controllers and actions.

By default, a rails application comes with this **l**ayout file
app/views/layouts/**application.html.erb**

You may find resource-specific layouts useful, but for now our app we will start with the single layout. Rails loads the layout file for each page you show, inserting the specific view page at the **<%= yield %>**. The cool thing is that you can add code into the layout, either directly or via a render, and it will show on all pages in your app.

Here is the default layout file, with a few additional lines:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inventory1</title>
```

```
    <%= stylesheet_link_tag    "application" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
        <%= render 'layouts/header' %>                [1]
        <%= yield %>
        <%= render 'layouts/footer' %>                [2]
        <% flash.each do |key, value| %>              [3]
              <br /><%= value %>
        <% end %>
        <%= debug(params) if Rails.env.development? %>  [4]
  </body>
  </html>
```

[1] Here we instruct Rails to insert this partial:
/app/views/layouts/_header.html.erb
A **partial** is like any other .html.erb file; it contains HTML and ruby code. It is a way to either isolate code that you don't want to clutter up the calling page (like here) or a way to reuse code in multiple pages. You can also call a render with arguments, but more later on that. Note that the leading underscore in the partial's filename does not appear in the call to render. I find this confusing but no one asked me.

[2] Same idea, different file.

[3] The flash is a globally-available hash of messages, keyed by type (notice, error, etc.). Once a flash message is shown, it won't be shown again. A controller typically loads messages into the flash to acknowledge errors and successes.

[4] To help us debug the app this will print the contents of the params hash on the screen on each page, but only when we are in the development environment.

5. Add Support for Users

Now we will generate a User model. You can add/change fields later via a migration but we might as well put in what we expect to need. (We'll need to add support for a superuser or admin, but lets defer that until later.)

```
rails generate model User encrypted_password:string
first_name:string last_name:string email:string
country_id:integer city:string province_id:integer
postal_code:string phone:string
```

Note that the model name User is **singular.**

This creates a model file, test files, and a migration. Lets focus on the first and the last.The model file, app/models/user.rb, consists of:

```
class User < ActiveRecord::Base
end
```

The migration, /db/migrate/YYYYMMDD####_create_users.rb, contains:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :encrypted_password
      t.string :first_name
      t.string :last_name
      t.string :email
      t.integer :country_id
      t.string :city
      t.integer :province_id
      t.string :postal_code
      t.string :phone

      t.timestamps                [1]
    end
  end
end
```

[1] Timestamp fields and an id field are automatically added by Rails.

The user model needs a fair amount of work to support all of the security that we need for our application. There are a lot of plugins that will serve up this functionality, but its also nice to see how its done. So we will roll our own code to support the following:

a. email field validation
b. password validation
c. reflect the user's relationship to other models (or we can do this later after we make those models)
d. password encryption
e. login authentication

Here is the completed code for the user model.

```
require 'digest'                        [1]
class User < ActiveRecord::Base

      # create a virtual password attribute
  attr_accessor :password                 [2]

  attr_accessible :email, :password, :password_confirmation, :first
  _name, :last_name, :country_id, :city, :province_id, :postal_code
  , :phone                      [3]
```

```
has_many :items                              [4]
email_regex = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i     [5]
validates :email, :presence => true,                    [6]
          :format => {:with => email_regex },
          :uniqueness => { :case_sensitive => false}

validates :password, :presence => true,                 [6]
                :confirmation => true,
                :length => { :within => 6..40 }

before_save :encrypt_password                          [7]

     # return true if the user's password matches the
     # submitted password

def has_password?(submitted_password)
          encrypted_password == encrypt(submitted_password)
end

def self.authenticate(email, submitted_password)
          user = find_by_email(email)
          return nil if user.nil?
          return user if user.has_password?(submitted_password)
end

     # method authenticate_with_salt()
     #
     # find the user by id
     # verify salt stored in cookie is correct for that user

def self.authenticate_with_salt(id, cookie_salt)
          user = find_by_id(id)
          (user && user.salt == cookie_salt) ? user : nil
end

private

def encrypt_password
          self.salt = make_salt if new_record?
          self.encrypted_password = encrypt(password)
end

def encrypt(string)
          secure_hash ("#{salt}--#{string}")
```

```
    end

    def make_salt
              secure_hash("#{Time.now.utc}--#{password}")
    end

    def secure_hash(string)
              Digest::SHA2.hexdigest(string)
    end
  end
```

[1] Pull in an encryption module
[2] attr_accessor defines setter and getter methods for each symbol passed as a parameter
[3] If a field isn't listed here you can' access it.
[4] Relationship between User and Item is 1:many.
[5] Regular expression for validating incoming emails
[6] Validation instructions
[7] Call this before saving to database

Edit the user controller so that the new method instantiates a user, and that the create method saves that user to the database. Add some status messages via the flash hash.

Create a form to edit information about a user. Use this form in two places, new user and edit user, with render. The one issue I have with this implementation is that it requires the user to re-enter password info.

Need to pursue: There seems to be an issue with validations and update_attributes.

6. Add Support for Sessions

For sessions we don't need a database, but we do need a controller and views. So generate a session sontroller.

```
rails generate Sessions new create destroy
```

Edit the session controller and session helper to support creation and deletion of sessions. We use the Sessions controller's module to make sessions helper functions available to all our controllers. To do this, include the SessionsHelper in applicationcontroller. Then write code to sign in, sign out, and get current user if there is one.

Here is the app/controllers/application_controller.rb (all controllers inherit from ApplicationController):

```
class ApplicationController < ActionController::Base
     protect_from_forgery
```

```
        include SessionsHelper
    end
```

We need a way to track each user from page to page, and it would be nice to have a persistent session -- one that lasts between browser sessions. So we'll generate a remember_token based on the user's salt and id and add a timestamp to the token, resetting thr token each time the user signs in.

Create the form for the user to login. It should be in app/views/session/new.html.erb and contain only email and password fields.

Clean up the config/routes.rb file. To do this, I removed the many routes added by the generate controller and used the simpler resources terminology. (There is more than one way to do routing, but I want to stick to the simplest.)

7. Cancelling a Form

I've found getting a nice cancel button on a form to be a bit of a challenge. I want it to go where I want AND to look like the submit button.

You can easily add a cancel **link** to go wherever you want:

```
<%= link_to "Cancel", :controller => , :action =>  … %>
```

but link_to doesn't have the same appearance as the submit button:

```
<%= f.submit "Save" %>
```

I solved the problem as follows. Add this to the form

```
<%= submit_tag "Cancel", { :name => "< model name >[Cancel]" } %>
```

Then check for the cancel text in the controller:

```
if !params[:< model name >].include?( 'Cancel' )
```

8. Add some business logic: the item model, views, and items_controller

Generate the item controller and views.

```
rails generate model item name:string description:text
user_id:integer number:integer tags:string location:string
category_id:integer status_id:integer

rails generate controller items new edit destroy show index
create update
```

Edit the routes in config/routes.rb to use the “resources :items” syntax.

> There are other ways to phrase routing, such as the “rails generate controller” ’s default routes, but I’m going to stick with what I think is the cleaner way. You need to make a choice as the named routes will vary depending on the technique.
>
> For example, “get ‘animals/show’ " gives a different named route than “resources :animals”. Its easy to test this, just edit config/routes.rb, save it , then run rake routes. Rake routes doesn’t look at your code to see if you actually have any controllers, it just looks at routes.rb.

Edit the controller to add code to support new, create, edit, destroy, and index methods. In the case of index and new, get the current logged in user from current_user.

Edit the User and Items models to reflect their associations.

```
class User < ActiveRecord::Base
        has_many :items
end

class Item < ActiveRecord::Base
        belongs_to :user
end
```

9. Lookup Tables for Inventory Items

I’ve specified two lookup tables, category and ownership. (The latter was originally named Status but I find that too vague and didn’t like the plural, Statuses.)

Category will contain a growing list of types of items we’d put in our inventory (“Furniture”, “Applicances”, etc.), whereas the latter contains a phrases that define the ownership status of an item, such as “Currently Own”, “Want”, “Want to sell”, “Want to sell/ trade”, and “Want to give away”.

So lets start by making the models:

```
rails generate model category name:string
rails generate model ownership name:string
```

Update our models to show their associations

```
class Item < ActiveRecord::Base
        belongs_to :user
        belongs_to :category
        belongs_to :ownership
end
```

```
class Ownership < ActiveRecord::Base
    has_many :items
end

class Category < ActiveRecord::Base
    has_many :items
end
```

Generate the database tables and the model

```
rake db:migrate
```

And make the controllers

```
rails generate controller categories new edit destroy show
index create update

rails generate controller ownerships new edit destroy show
index create update
```

Next I want to seed the lookup tables with data. But note that the controllers will support editingb the table contents, too, though I'll limit access to the admin user. Which reminds me we'll need to add a way for a user to have admin status. More on this later

Once we make the lookup tables, we can seed them. Since this is typically something you may want to repeat, make a seed file for each table that gets seeds, naming it someting like db/<model>_seeds.rb. Each line of the seed file creates a record in the table:

```
<model>.create( :<fieldname> => <value>, :<fieldname> => <value>, ... )
```

Then copy or rename the file db/seeds.rb and run the following

```
rake db:seed
```

Check that the table is now populated with data. Do this for both the Categories table and the Ownerships table.

Since I changed the name of the model from status to ownership, I need to drop and field and add a new field in the item model.

```
rails generate migration add_ownership_id_to_item
```

Before I run the migration, edit this migration to add a field and remove a field. . db/YYYYMMDD####_add_ownership_id_to_item.rb is as follows:

```
class AddOwnershipIdToItem < ActiveRecord::Migration
  def change
    add_column :items, :ownership_id, :integer
    remove_column :items, :status_id
  end
end
```

Then run

```
rake db:migrate
```

Now we want to use the lookup tables in the New and Edit items forms. To do this we need to collect values for the lookup tables and pass them to the view via an instance variable (which starts with an '@' sign). There are two aspects to this: you need a query of the database and a way to show a dropdown in the form. First, we like to mode as much of the knowledge about the model onto the model so place the query of the lookup table into the model, not the controller, like so:

```
class Category < ActiveRecord::Base
    has_many :items
    scope :ordered, order("name")
end
```

A scope is a shortcut for defining a method on a model. You could also make an explicit method, but this is a bit cleaner. To call this method just add this to your controller:

```
@categories = Category.ordered
```

Second, in the form you need to access this array of categories, and to do so we use a collection_select. The first time I used collection_select (in Rails 2.3.8) it took a lot of searchig to get it to work. More info here: http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html#method-i-collection_select

```
<%= f.collection_select :category_id, @categories, :id, :name, :prompt => "Select Category" %>
```

Repeat this for the Ownership model.

10. Lookup tables for Users

To complete the User form fields we need lookup tables for country and province (which includes state). Here are the model generators:

```
rails generate model Country name:string
```

```
rails generate model Province name:string abbreviation:string country_id:integer
```

Now run

```
rake db:migrate
```

The dropdown for Province will include all provinces and states. We won't need a controller for either as we'll populate them from seed data.

The seed file for countries contains:

```
Country.create(:name=>"Canada")
Country.create(:name=>"United States")
```

The seeds file for provinces looks something like this:

```
Province.create(:name=>"Alberta", :abbreviation=>"AB", :country_id=>1)

Province.create(:name=>"British
Columbia", :abbreviation=>"BC", :country_id=>1)

Province.create(:name=>"Manitoba", :abbreviation=>"MB", :country_id=>1
)

Province.create(:name=>"New
Brunswick", :abbreviation=>"NB", :country_id=>1)
  …
```

Now seed each table :

```
rake db:seed
```

Edit the models (app/models/...) to reflect the relationships between User and Province and Province and Country. Later we'll see how we can access Country through Province.

11. Change Link_to to button_to for delete

In app/views/items/index.html.erb I replaced ***link_to "delete"*** with ***button_to "delete",*** which fixed the problem I had with the delete method not working. I've learned that link_to shouldn't be used for deletions. Also removed the remove method in the items controller and use delete (or is it destroy?) instead, and removed the call to remove in routes.rb.

12. Add some css

I have two goals for this section: make the UI a little nicer looking and make the menu items sensitive to context. In the case of the former, I revised the names of menu items and where they are placed and where they go. But before we go into the implementation, this is how I want the menus to work.

If the user has not signed in, this is the top-left menu:

- My Inventory - greeting / describes the app
- Sign In - form
- Sign Up - form

Else if user has signed in

the top-left menu

- My Inventory - list of current inventory: name, category,links to edit and delete
- Search - coming soon
- Help - ditto

the top right menu

- Sign out
- (email) - edit profile form

Across the bottom is always:

- About - static

Contact - static

As our app grows, we may decide to break styles into separate files for style scopes. But I want to start simple, so lets start out by placing all our styles in an application-wide style file.

app/assets/stylesheets/**application.css**

I've also edited the three files in

app/views/layouts/**application.html.erb**
app/views/layouts/**_header.html.erb**
app/views/layouts/**_footer.html.erb**

Plus the controllers for users and items.

app/controllers/**users_controller.rb**
app/controllers/**items_controller.rb**

I'm not wedded to this implementation but it demonstrates the desired behaviour.

13. Add support for paid and admin user

A simple scheme for categorizing users might be basic (0), paid (1), or admin (2).

admin - can view users, edit their information, edit their items, and edit lookup tables for category and ownership. Can also search across all items and see who owns it.

basic - can view and edit owned items. Can search on items but can see limited info on who owns it.

paid - ??

everyone else - must sign in or sign up to get beyond home, about, and contact

When a new user is created they are of type 0. If a paid user, type 1, admin would be type 2. Start by adding a field in the user model:.

```
$ rails generate migration add_usertype_to_user  user_type:integer
      invoke  active_record
      create    db/migrate/20111121014405_add_usertype_to_user.rb
```

Lets look at db / migration / **201111121014405_add_usertype_to_user.rb**

```
class AddUsertypeToUser < ActiveRecord::Migration
  def change
    add_column :users, :user_type, :integer
  end
end
```

It seems that Rails 3 no longer automatically adds a method to reverse the change (perhaps there is an argument to generate to change this?).

```
$ rake db:migrate
==  AddUsertypeToUser: migrating
=============================================
-- add_column(:users, :user_type, :integer)
   -> 0.0005s
==  AddUsertypeToUser: migrated (0.0007s)
 ====================================
```

Edit the model file, app/models/user.rb to make the new field accessible and to require that it be set when a user is saved

Now, edit users_controller#new to init value to 0. And add a field in the users_controller#edit form so that the admin can edit the new field. .

I'm going to run the migration, then change the users@new method to initialize this to 0, and I'll change the sessions_helper#admin? to check for this field == 2 . Finally, i'll show it on the users#index page.

I want to add more to the users#index page Add a column showing how many items they own, and maybe we'll show a graph later to show types of items.

As I test my links on the admin page, I discover that there is no provinces controller, so we'll make one:

```
$ rails generate controller Provinces show edit new create update
delete index
      create  app/controllers/provinces_controller.rb
       route  get "provinces/index"
       route  get "provinces/delete"
       route  get "provinces/update"
       route  get "provinces/create"
       route  get "provinces/new"
       route  get "provinces/edit"
       route  get "provinces/show"
      invoke  erb
      create    app/views/provinces
      create    app/views/provinces/show.html.erb
      create    app/views/provinces/edit.html.erb
      create    app/views/provinces/new.html.erb
      create    app/views/provinces/create.html.erb
      create    app/views/provinces/update.html.erb
      create    app/views/provinces/delete.html.erb
      create    app/views/provinces/index.html.erb
      invoke  test_unit
      create    test/functional/provinces_controller_test.rb
      invoke  helper
      create    app/helpers/provinces_helper.rb
      invoke    test_unit
      create      test/unit/helpers/provinces_helper_test.rb
```

```
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/provinces.js.coffee
      invoke    scss
      create      app/assets/stylesheets/provinces.css.scss
```

One thing I've discovered is I should have used Categories for Categorys …

```
$ rails generate controller Categories show edit new create update
delete index
```

14. Change admin implementation

If you haven't studied the last chaper, read this chapter first. The way we stored admin status in the last chapter strikes me as, well, boneheaded. Lets change it to a boolean that strictly indicates admin status, nothing more. And we'll replace the field added earlier, user_type:integer, to be an index into a lookup table of user_types.

```
$ rails generate migration add_admin_to_user admin:boolean
      invoke  active_record
      create    db/migrate/20111122232218_add_admin_to_user.rb

$ rails generate migration drop_usertype_in_user
      invoke  active_record
      create    db/migrate/20111122232236_drop_usertype_in_user.rb
```

Lets look at the migrations.

I added one line to the second because the generator didn't know what I wanted to do.

So if the generator knows what you want -- for example, the descriptive word "add" plus specs for a new field -- it will use the change method. Otherwise it provides the up and down methods but you must fill them in.

```
class AddAdminToUser < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean
  end
end

class DropUsertypeInUser < ActiveRecord::Migration
  def up
      remove_column :users, :user_type
  end

  def down
  end
end
```

```
$ rake db:migrate
==  AddAdminToUser: migrating
=============================================
-- add_column(:users, :admin, :boolean)
   -> 0.0004s
==  AddAdminToUser: migrated (0.0006s) ========================================

==  DropUsertypeInUser: migrating ==========================================
-- remove_column(:users, :user_type)
   -> 0.0410s


==  DropUsertypeInUser: migrated (0.0412s) =================================
```

15. Add ability to edit lookups and add new ones.

Users with admin rights see a few additional links on the lower right corner of the page. These link to controllers that allow the editing of categories, ownerships, and users. Also, the inventory index page shows ALL items for this user.

To do: admin should be able to narrow items to owned items.

The lookups were implemented by simply addin controller and vies for each. One thing we might want to do is limit access to users who are admins.