How to Create the Inventory App on Rails 3.1.1

Here is an outline for developing a typical web app. Clearly, your app's needs may differ but this app will show a lot of the basics that most any app will need.

As I develop the code I'll push it out to github, under my 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 may exist, but coming up with something revolutionary is beyond the scope of this course :-)

The strategy of attacking this problem is as follows:

- Plan the app models
- Create the app and common static pages
- Add support for users (new user, user login, user logout, edit user profile) and Sessions (login, logout, persistent connection)
- Add business logic
    - Add support for our inventory items: new, edit, delete, index
    - Add lookup tables for inventory item categories and ownership status
- Add lookup tables to link to the User model (country, province/state)
- Add support for Users with admin status so they can edit lookup tables, etc.
- Add social support: allow users to search for users to sell/trade/aquire stuff
- Enhance the Items table: perhaps we need size, condition, value, and/or other fields? Or links to item-specific fields?
- Improve the UI wth css, images, jQuery, Ajax
- Etc.

Plan the App Models

Most apps need some **static pages**, such as about, help, signin, signout, contact, etc. For our app I'll not be storing anything in a database for these pages -- your needs may vary.

Most apps need to support **users**, and need to store, at a minimum, a user's email address and encrypted password. Here you should identify other fields you may want to store in this model, such as name, location, last login, etc. Alternatively, your users may vary in type of information needed in which case you'd have separate model(s) for different types of user and a foreign key into the users table. For the inventory app, all users are the same so I'll put everything in the user model.

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

Identfy data needs for your business logic. For the inventory app, I'll have at a minimum an items table plus a growing set of lookup tables.

Create the App and Common Static Pages

Generate the 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 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'** ".

Install the gems that are specified by Gemfile, then write a gem snapshot to Gemfile.lock.
I'm happy with the defaults 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

Generate controller for typical static pages: home, about, contact, help, signup, signin.

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

Look at the Pages controller (app/controllers/pages_controller.rb). For the moment I'll leave it as is since all it needs to do is pass control from a controller method to a view. But note that there is a controller method for each view we specified on the generate controller command line. And note that sometimes we don't need all the views generated because we may redirect in the controller to some other action instead.

Edit the home page (app/views/pages/home.html.erb) so it says something of interest

Remove public/index.html. We'll let the router control what is loaded.

Check the generated routes in config/routes.db so that they route to the static pages. We'll talk more about routes later ...

Create header and footer partials in the app/views.layouts directory

Edit app/views/layouts/application.html.erb so that it renders the header and footer. Note that the view is inserted at the <%= yield %>.

Add Support for Users and Sessions

Generate User model specifying fields. You can add/change fields later via a migration but we might as well put in what we know.

```
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
```

Add code to the User model (app/models/user.rb) to support

a. email 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

Edit the user controller so that the new method instantiates a user, and that the create method saves that user to the database. We'll also add some status messgaes via the flash hash.

Edit the new user view (app/views/user/new.html.erb) to have a form for the user to enter basic user information.

As an aside, I've found getting a cancel button on a form to go where I want to be a surprising challenge. You can add a cancel link to go wherever you want easily:

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

but this 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 => "user[Cancel]" } %>
```

Then check for the cancel text in the controller:

```
if !params[:user].include?( 'Cancel' )
```

Contining on, lets generate a session sontroller.

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

Write the session controller and session helper code to support creation and deletion of sessions. We'll also need to edit the User model a little more ...

f. 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 based on the user's salt and id and add a timestamp to the token, resetting thr token each time the user signs in.
g. We are going to 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.

Create the form for the user to login. It should be in app/views/session/new.html.erb and contain 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 wat to do routing, but I want to stick to the simplest for now.

Add an abiliy to edit the logged in user's profile. This means adding a link in the header partial, adding a form to app/views/users/edit.html.erb (you can re-use the one in new and in fact move this form to a partial and have both access it via render), and then add code to the users_controller to find the user record and then update it once it is saved.

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
```

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, like so:

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

The first time I used collection_select (in Rails 2.3.8) it took a lot of searchig to get it to work.