



[home](#)
[articles](#)
[quick answers](#)
[discussions](#)
[features](#)
[community](#)
[help](#)

Search for articles, questions, tips



Articles » Web Development » Applications & Tools » Applications



Article

[Browse Code](#)

[Stats](#)

[Revisions](#)

[Alternatives](#)

[Comments \(15\)](#)

User Authentication in Ruby on Rails



Marc Clifton, 10 Apr 2013

★★★★★ 5.00 (15 votes)

Rate this:



A comprehensive look at getting authentication working for Rails applications

Contents

- [Introduction](#)
- [Installing the Basics](#)
 - [Installing Ruby on Rails for Windows](#)
 - [Installing PostgreSQL](#)
 - [Installing RubyMine](#)
 - [Additional Nice Things](#)
 - [Ansicon](#)
 - [Firefox](#)
 - [SmartGit](#)
- [Getting a Basic Rails Website Running](#)
 - [Testing the Website](#)
 - [Stopping the Server](#)
 - [Sometimes You May Need to Manually Clear the Server's PID](#)
- [Create The Database](#)
 - [Inspect the database.yml File](#)
 - [Create the User Role in PostgreSQL](#)
 - [Run the Rake Command to Create the Database](#)
- [Setting up Source Control](#)
 - [Create a GitHub Account](#)
 - [Configure GitHub for RubyMine](#)
 - [Share the Project on GitHub](#)
- [Learning Your Way Around a Rails Project](#)
 - [Models, Views, Controllers and Tests](#)
 - [Gemfile, Migrations and Routes](#)
- [Creating the Database User Table as a Migration](#)
 - [Run the Migration](#)
- [Creating the User Model](#)
 - [Encryption](#)
 - [Bundler](#)
 - [The User Model Code](#)
 - [Working with the RubyMine Rails Console](#)
 - [Authenticating by Username](#)
- [Controllers, Pass 1](#)
- [Views: The Authentication Pages](#)
 - [Remove the Default Index Page](#)
 - [Routes](#)
 - [The Root Route](#)
 - [The Authentication Routes](#)
- [The Sign In Page: Controller, View, and Routes](#)
 - [The View](#)
 - [The Route](#)
 - [The Controller](#)

About Article

Type	Article
Licence	CPOL
First Posted	10 Apr 2013
Views	57,960
Bookmarked	24 times

☐ Windows
 ☐ Dev
 ☐ Intermediate
 ☐ Ruby

Related Articles

[What is Ruby on Rails? \(An introduction\)](#)

[Install Rails 4 and MySQL Server on Windows 7/8](#)

[Introducing Castle - Part II](#)

[Where In The World Are My Facebook Friends?](#)

[Day 1: Spider Database Navigator Website](#)

[Ruby on Rails on Windows in production](#)

[Salted Password Hashing - Doing it Right](#)

[A Coder Interview With Marc Clifton](#)

[SOHA - Service Oriented HTML Application \(Session and Security\)](#)

[ASP.NET and Comet: Bringing Sockets Back](#)

[A Coder Interview With Maureen McElaney](#)

[ASP.NET authentication and authorization](#)

[Tom & Jim Podcast #1: The Code Project Article](#)

[SOHA - Service Oriented HTML Application \(Concepts and Principles\)](#)

[Single Sign On \(SSO\) for cross-domain ASP.NET applications: Part-I - The design blue print](#)

[Comparing Ruby and C# Performance](#)

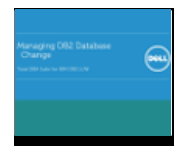
[Authentication and Authorization in ASP.NET](#)

[Simple Introduction About What is Ruby on Rails](#)

[A Coder Interview With Vanessa Hurst](#)

[ASP .NET MVC for Beginners in Web Development](#)

- ## Related Research



How to Apply Changes to Your DB2 Database with Minimal Risk



Toad Oracle: Discover the Top 10 Most Useful Toad Features



Data Modeling for Effective Data Warehousing and Business Intelligence



Toad Oracle: Tips to Simplify Database Administration and Development

- Sign Up
- Sign In
- Sign Out
- Account Management
- Password Reset
- Basic User Administration

Exploring such features as:

- Gems
- Site Environment Variables
- Database Migrations
- HTML metadata
- Controllers, Models, and Views
- Application Layouts
- CSS
- Cookies

Also, creating a basic authentication system from which you can then build on for other website features is an essential requirement for most websites and provides a good basis for this tutorial.

If you are new to Ruby and familiar with C#, you may want to first read my article comparing [C# and Ruby Classes](#) to become more familiar with Ruby syntax.

Installing the Basics

The first thing that needs to be done is to install a few things:

- The Ruby and Rails runtime for Windows
- PostgreSQL, which we will be using as the back-end database
- A decent editor, such as RubyMine

Once this is complete, we can create the scaffolding for a Rails website and the authentication forms.

Installing Ruby on Rails for Windows

First, go to the website <http://railsinstaller.org/> and download and install the kit for Windows. As the page says, "RailsInstaller has everything you need to hit the ground running" and his highly recommended!

Installing PostgreSQL

RailsInstaller comes with Sqlite, however we're going to want to use a more "professional" database, and if you want to push your application onto a virtual host such as [Heroku](#), I highly recommend that you use PostgreSQL. The Windows [download for PostgreSQL](#) includes the database server and a decent Windows app for managing the database called pgAdmin III. It also provides the option to install StackBuilder, which you can decline.

Installing RubyMine

RubyMine by JetBrains is an excellent IDE. I recommend [downloading the trial version](#). The IDE provides an excellent editor, integrated source control management, debugging, and tool support. With RubyMine, one hardly ever needs to open up a command-line prompt, which I prefer to avoid myself.

Additional Nice Things

Ansicon

For times when it is useful or necessary to use the command line, I recommend installing [Ansicon](#), a console app that knows how to process ANSI escape sequences, which are used for formatting and colorization of many of the useful things one can do from the command line.

Firefox

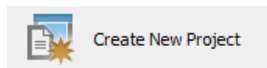
I also recommend installing [Firefox](#), as the browser engine is used for certain integration test modes. Unit testing and integration testing is something I will cover in the next article.

SmartGit

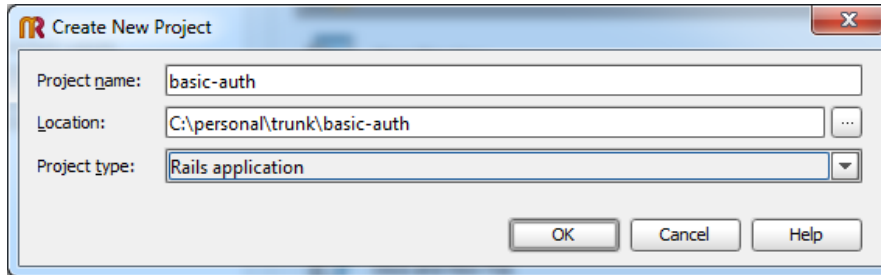
This is a Windows wrapper for the source control system Git and is an indispensable tool for eliminating the confusion that beginners (and experienced users of other version control systems!) will otherwise encounter with the command line operation of Git. However, because RubyMine integrates version control (SVN and Git being two of the several options), we will focus on using version control within RubyMine.

Getting a Basic Rails Website Running

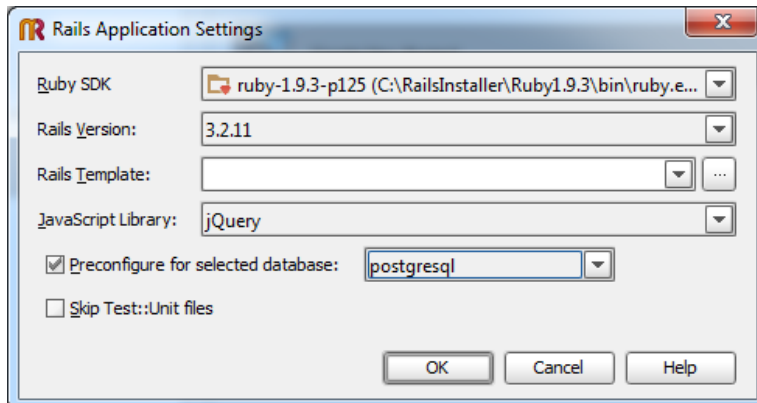
Once you have the various programs described above installed, launch Ruby-Mine and select "Create New Project":



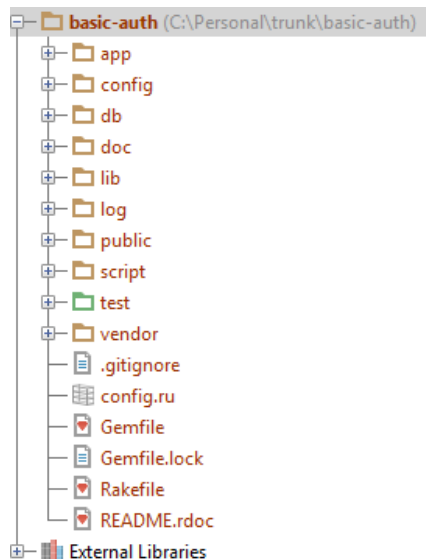
Enter in the project name, location, and select "Rails application" for the project type:



On the second screen "Rails Application Settings" change the "selected database" to "postgresql":



RubyMine will create the scaffolding for your website:



Much of what are in these folders will be touched upon in this article, so hang on to your hats!

Testing the Website

At this point, you can test the website that RubyMine created. From the menu, select Run and from the dropdown, "Run 'development: basic-auth'". This will launch the WEBrick application server. Once it is initialized:

```

Run Development: basic-auth
Console Server development log x
runnerw.exe C:\RailsInstaller\Ruby1.9.3\bin\ruby.exe -e $stdout.sync=true;$st
=> Booting WEBrick
=> Rails 3.2.11 application starting in development on http://127.0.0.1:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2013-04-06 15:30:08] INFO WEBrick 1.3.1
[2013-04-06 15:30:08] INFO ruby 1.9.3 (2012-02-16) [i386-mingw32]
[2013-04-06 15:30:08] INFO WEBrick::HTTPServer#start: pid=4724 port=3000

```

open up your favorite browser and, for the URL (except Internet Explorer), enter "localhost:3000". If you are using Internet Explorer, enter "<http://localhost:3000/>" - the "http://" is essential, otherwise Internet Explorer (at least this is how Version 9 behaves) will try to do a search using Bing.

You will then see the welcome screen (a portion of which is in this screenshot):

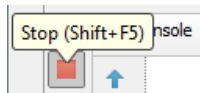
Welcome aboard

You're riding Ruby on Rails!

[About your application's environment](#)

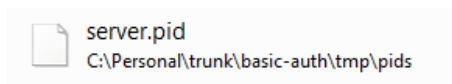
Stopping the Server

To stop the web server, click on the red square:



Sometimes You May Need to Manually Clear the Server's PID

I have on occasion experienced an improper shutdown of the server, especially when debugging, which requires manually deleting the PID file, which is located in tmp\pids from the project folder. For example, when the server is running, here is the PID file:



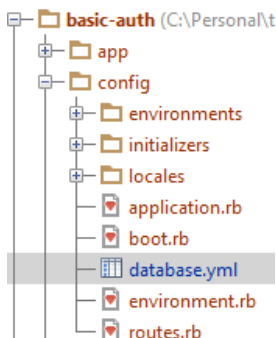
If RubyMine reports that a server instance is already running, find your PID file and delete it, then try starting the server again.

Create The Database

There are several ways to do this, but for our purposes I am going to demonstrate how this is done (mostly) from within RubyMine.

Inspect the database.yml File

First, open the database.yml file in RubyMine, found in the config folder:



A YAML file is a bit like the (no mostly obsolete) INI files that Windows applications used to use. It is essentially comprised of sections and key-value pairs, with some additional configuration features. Note the "development:" section and that the username is the same as the application name:

```
development:
  adapter: postgresql
  encoding: unicode
  database: basic-auth_development
  pool: 5
  username: basic-auth
  password:
```

If you have an existing PostgreSQL installation, you can replace the username and password with an existing username and password, however, for this tutorial, I will assume that you are new to PostgreSQL. Edit the yml file to specify a password, for example:

```
development:
  adapter: postgresql
  encoding: unicode
  database: basic-auth_development
  pool: 5
  username: basic-auth
  password: mypassword
```

Do the same with the other two sections:

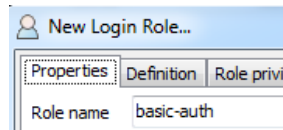
- test
- production

This way, the step below, which actually creates the databases, will successfully create the development, production, and test databases. And this is an important point to note, that Ruby on Rails and the RubyMine IDE have sophisticated integrated features for working in a development environment, migrating changes to the production environment, and also creating unit and integration testing in a test database. Furthermore, since these are separate sections, each of the three databases can utilize a different database engine. Sqlite, for example, is commonly used as the test database.

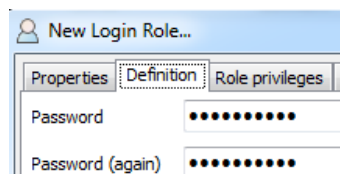
Create the User Role in PostgreSQL

Open the pgAdmin III application (found from the Start menu under PostgreSQL 9.2 (or later version)). Double click on the "PostgreSQL 9.2" item in the tree under "Servers" and right-click on Login Roles.

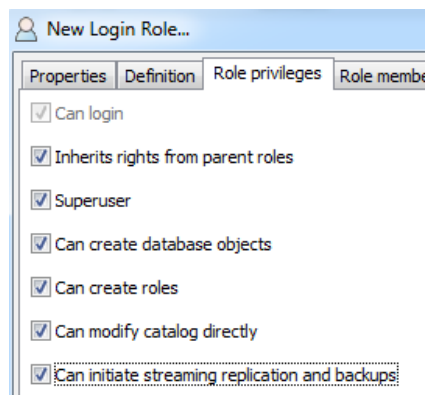
In the first tab, "Properties", enter the role name, in this case, "basic-auth":



Click on the "Definition" tab and enter the password, in this case, "mypassword" in both fields "Password" and "Password (again)":



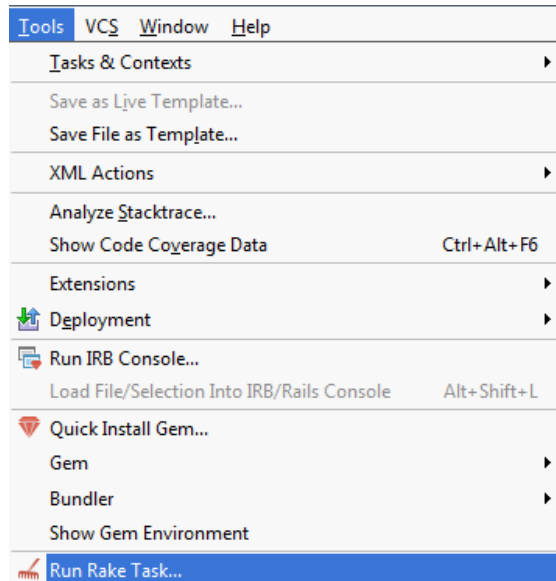
Click on the "Role privileges" tab and select all the checkboxes, as we want to give this user all permissions:



Click on OK to create the role.

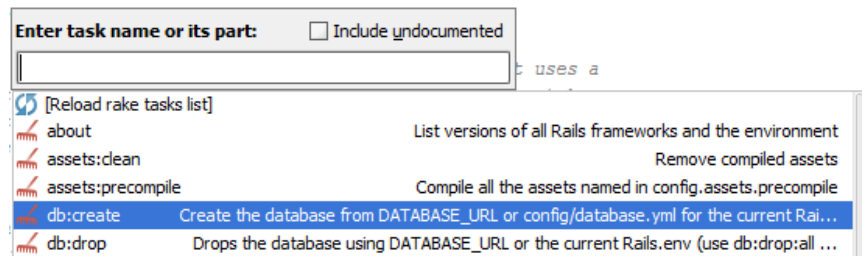
Run the Rake Command to Create the Database

Back in RubyMine, from the Tools menu select "Run Rake Task...":

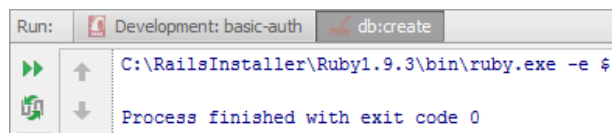


Rake is an application that does a lot of tasks related to creating and maintaining the state of your application. Most of the features of rake will not be covered in this tutorial!

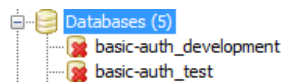
A dialog is presented, from which you should select the db:create task:



If the role has been set up correctly, the console output from RubyMine will show:



To confirm that the databases have been created, right-click in the pgAdmin tool the "Databases" tree node, click on "Refresh", and note that the development and test databases have been created:



Note that the rake db:create command does not create the production database. Also note that the red X simply means that we haven't connected to the database in the pgAdmin tool.

Setting up Source Control

Before we get too much further, it would be useful to set up source control using Git, which is the popular source control repository system for Rails projects.

Create a GitHub Account

A simple way to create a remote repository is to use [GitHub](#). Create an account in GitHub. RubyMine will automatically create a repository (see "Create the Repository" below.)

Configure GitHub for RubyMine

From the File menu, select Settings, then open the Version Control node and select GitHub:

Enter the login and password for the GitHub account created above, for example:

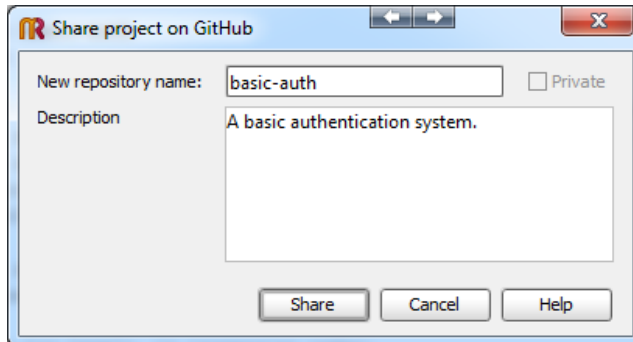


Very Important: For the login, specify your *username*, not your email address. While GitHub allows you to login with either your username or your email address, authentication for the repository itself requires your username.

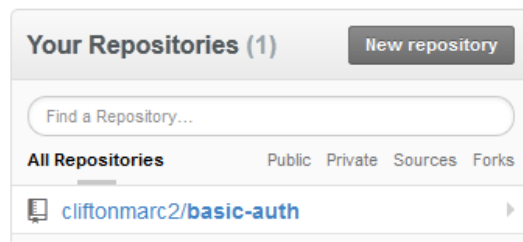
Share the Project on GitHub

In RubyMine, click on the VCS main menu, then "Import Into Version Control", then select "Share Project on GitHub".

You will be prompted for a description of your project:



Click on Share. Assuming your login credentials (above) were entered correctly, the repository will be automatically created and the initial contents pushed to the remote repository. You can verify this on the GitHub website - login (or if you are already logged in, navigate back to the GitHub main page) and you should see:



The project is now configured for using GitHub as the remote repository!

Learning Your Way Around a Rails Project

Working with Rails means that you are most likely initially going to fuss with:

- gems
- migrations

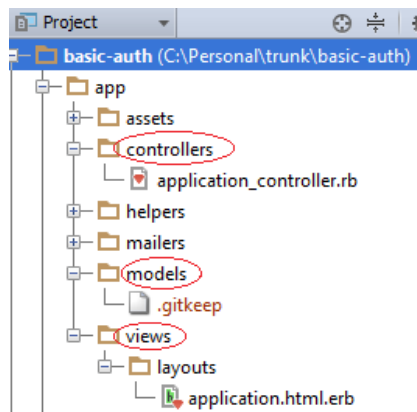
and then once the plugins and database model is fairly stable, you will find yourself working more with:

- models
- views
- controllers
- routes

I should mention also tests (in the "test" folder), but that will be covered in the next article.

Models, Views, Controllers and Tests

A Rails project maintains the files for the models, views, and controllers under the "app" folder:



Note that by default, there is an "application_controller.rb" Ruby file as well as an "application.html.erb" view, the former being the controller for the latter. The default application view is where you specify all the common visual elements of your website.

If we take a quick peek inside application.html.erb, we find this important piece of code:

```
<body>

<%= yield %>

</body>
```

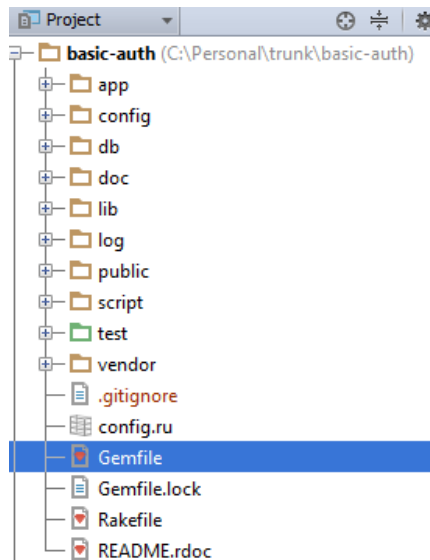
[Collapse](#) | [Copy Code](#)

The "yield" statement yields control to the view specific to the page that is being rendered. Anything common features of your website, such as a title, menubar, copyright notice, etc., can go before and after the yield statement. The behind-the-scenes implementation of what exactly yield ends up doing is (among many other things) what Rails provides.

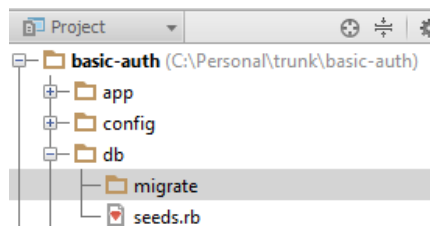
Gemfile, Migrations and Routes

An initial Rails project usually requires that you add plugins (gems) into the "gemfile", work with database migrations as you add physical table structures to your database, and configure the routes to your web pages.

The gemfile is located in the root of your project folder:

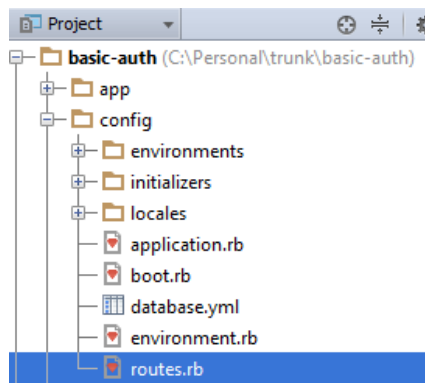


The database migrations are usually kept in a "migrate" folder under the "db" folder:



At this point, create the "migrate" folder - it is not created automatically for you.

The routes.rb file is in the config folder:

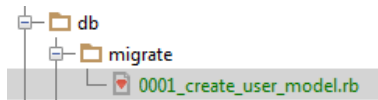


As you can see, there's a lot to keep track of, and I've only presented the bare minimum at this point.

Creating the Database User Table as a Migration

Migrations are a good way of creating the database model for a given version of your website. You can create and alter tables as well as revert to previous versions of the model. While one can create a model from the command line using the "rails g model..." command, I'm going to try and avoid the command line to illustrate how to work with migrations manually rather than from generated code. The advantage to this approach is that you will learn more of the inner workings of Rails, which is important when faced with making changes that the command line generators do not support.

A migration file has the following features:



First, the filename begins with a developer-created index, either an incremental value or a date-time stamp. This is followed by the migration name as underscore separated words. This is important, because the Ruby migration class must match this pattern, but without the underscores and is also Pascal-cased:

```
class CreateUserModel < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :username, :string
      t.column :email, :string
      t.column :password_hash, :string
      t.column :password_salt, :string
    end
  end

  def self.down
    drop_table :users
  end
end
```

[Collapse](#) | [Copy Code](#)

Note the Pascal-cased class name "CreateUserModel". Also note that the class is derived from "ActiveRecord::Migration", which is provided by Rails.

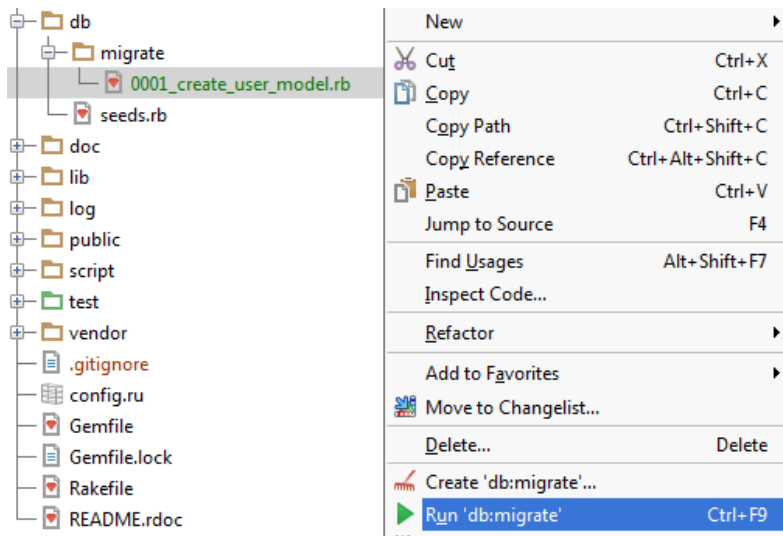
A migration file typically consists of two functions:

- self.up - this function describes the changes being made to the model.
- self.down - this function describes how to revert back to the previous state.

Rails provides a rich set of functions for creating and altering tables. Only two functions, "create_table" and "drop_table" are illustrated here. Another very common feature of Rails applications is that they are very *symbol* oriented. Note that instead of using quoted strings for the table and field names, we are passing in symbols, which are always preceded by a colon (:). Note that symbols are also used for the field types.

Run the Migration

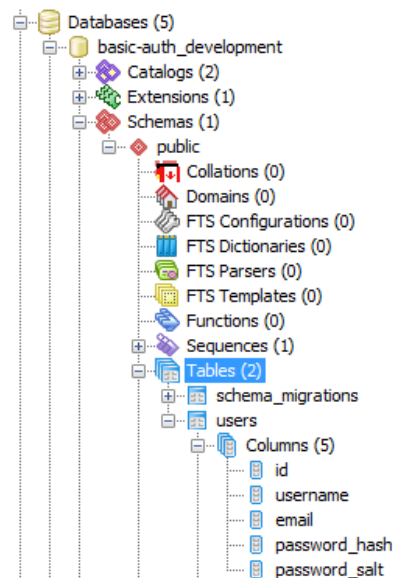
Once the migration file has been created, right-click on the migration file and select "Run 'db:migrate'":



If the migration is successful, the output view in RubyMine will display something like this:

```
Run db:migrate
C:\RailsInstaller\Ruby1.9.3\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load($0=ARGV.shift)
NOTICE: CREATE TABLE will create implicit sequence "users_id_seq" for serial column "users.id"
== CreateUserModel: migrating =====
-- create_table(:users)
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "users_pkey" for table "users"
-> 0.0160s
== CreateUserModel: migrated (0.0160s) =====
Process finished with exit code 0
```

Note that the primary key is automatically added by the migration code, as well as creating a backing sequence to generate the primary key for when new rows are added. You can inspect the table in pgAdmin III:



Note the new "users" table. Rails also creates a "schema_migrations" table to keep track of the migrations that have already been run. Do not edit this table.

Creating the User Model

Once the database table has been created, the Rails user model can be created. Under the "models" folder, create "user.rb". Some things of note:

1. Note that the model class names that back the respective tables are singular, while the table names are plural. This convention is enforced by Rails.
2. Note that classes, being types, begin with a capital letter.
3. Models which are backed by physical tables are always derived from "ActiveRecord::Base", provided by Rails.

Encryption

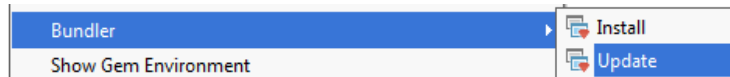
Obviously, we will want to encrypt the password in the database. For this, we will use the "bcrypt-ruby" gem, which means adding the following line to the gemfile:

```
gem 'bcrypt-ruby', :require=>'bcrypt'
```

Bcrypt is a plugin that provides basic hashing algorithm to encrypt strings.

Bundler

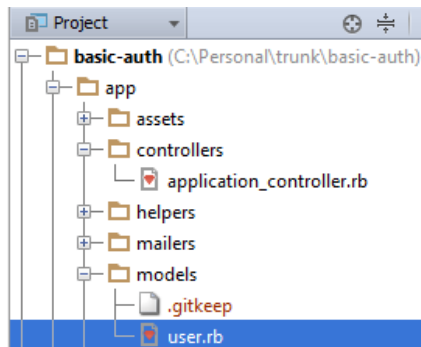
Any time a gem is added or removed to/from the gemfile, you should run the "bundler" installer or updater. The installer will install any missing gems, and updater will install missing gems and download updates if there are any to existing gems. To do this from RubyMine, select "Tools" from the main menu, then either Install or Update:



This will update the gems installed on your system.

The User Model Code

Create the user model in the "models" folder:



Note that the name for model filenames is the *singular* for the backing table name.

A basic implementation for the user model looks like this:

```
class User < ActiveRecord::Base
  attr_accessible :email, :username, :password, :password_confirmation
  attr_accessor :password
  before_save :encrypt_password

  validates_confirmation_of :password
  validates_presence_of :password, :on => :create
  validates_presence_of :email, :on => :create
  validates_presence_of :username, :on => :create
  validates_uniqueness_of :email
  validates_uniqueness_of :username

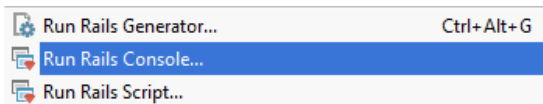
  def self.authenticate(email, password)
    user = find_by_email(email)
    if user && user.password_hash == BCrypt::Engine.hash_secret(password,
      user.password_salt)
      user
    else
      nil
    end
  end

  def encrypt_password
    if password.present?
      self.password_salt = BCrypt::Engine.generate_salt
      self.password_hash = BCrypt::Engine.hash_secret(password, password_salt)
    end
  end
end
```

Here we have a considerable amount of metadata that describes the validation of the model data, when to perform the validation, and code (`encrypt_password`) that should be executed before the model is saved to the database. I am not going to describe the code in detail - this code comes from the [Railscast #250](#). Later on we will modify it, but for now it's a good starting point. Incidentally, Railscasts is an outstanding resource for learning how to do things with Ruby on Rails.

Working with the RubyMine Rails Console

At this point, we can test the User class from the Rails Console. In RubyMine, select Tools from the main menu, then "Run Rails Console":



We can then interact with our User model, creating a user:

```
Run Rails console: basic-auth
C:\RailsInstaller\Ruby1.9.3\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load($0=ARGV.shift) C:/Personal
>> u = User.new
Loading development environment (Rails 3.2.11)
Switch to inspect mode.
#<User id: nil, username: nil, email: nil, password_hash: nil, password_salt: nil>

>> u.username='cliftonm'
"cliftonm"

>> u.email = 'marc.clifton@outlook.com'
"marc.clifton@outlook.com"

>> u.password='mypassword'
"mypassword"

>> u.password_confirmation = 'mypassword'
"mypassword"

>> u.save
(0.0ms) BEGIN
User Exists (3.0ms) SELECT 1 AS one FROM "users" WHERE "users"."email" = 'marc.clifton@outlook.com' LIMIT 1
User Exists (0.0ms) SELECT 1 AS one FROM "users" WHERE "users"."username" = 'cliftonm' LIMIT 1
SQL (1.0ms) INSERT INTO "users" ("email", "password_hash", "password_salt", "username") VALUES ($1, $2, $3, $4)
(4.0ms) COMMIT
true
```

and we can also verify that we can find users using the "static" authenticate method:

```
>> User.authenticate 'marc.clifton@outlook.com', 'mypassword'
User Load (1.0ms) SELECT "users".* FROM "users" WHERE "users"."email" = 'marc.clifton@outlook.com' LIMIT 1
#<User id: 1, username: "cliftonm", email: "marc.clifton@outlook.com", password_hash: "$2a$10$GyG5HYD1HM9ZmYup", password_salt: "cliftonm">

>> User.authenticate 'marc.clifton@outlook.com', 'badpassword'
User Load (1.0ms) SELECT "users".* FROM "users" WHERE "users"."email" = 'marc.clifton@outlook.com' LIMIT 1
nil
```

Note how the first call with the correct password returns a User record, while the second call, with an incorrect password, returns a nil.

Authenticating by Username

Now let's add a method to authenticate with the username, as well as renaming the methods a little bit to be clearer as to how what fields we are using to authenticate the user:

[Collapse](#) | [Copy Code](#)

```
def self.authenticate_by_email(email, password)
  user = find_by_email(email)
  if user && user.password_hash == BCrypt::Engine.hash_secret(password,
    user.password_salt)
    user
  else
    nil
  end
end

def self.authenticate_by_username(username, password)
  user = find_by_username(username)
  if user && user.password_hash == BCrypt::Engine.hash_secret(password,
    user.password_salt)
    user
  else
    nil
  end
end
```

If you were to test these new methods in the Rails Console, you will have to restart the console.

We can verify that the user has been created in the database by using pgAdmin III to inspect the table:

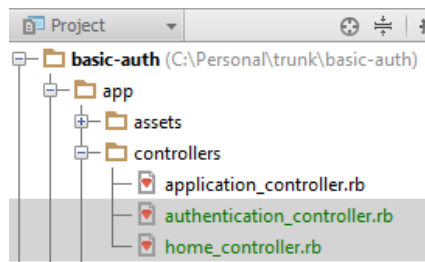
	id [PK] serial	username character va	email character va	password_ha character va	password_sa character va
1	1	cliftonm	marc.clifton@outlook.com	\$2a\$10\$GyG5HYD1HM9ZmYup	cliftonm
*					

Controllers, Pass 1

Controllers are an integral part of Rails applications - you must have a controller that is associated with views. To begin with, we'll create two controllers as stubs:

1. home_controller.rb
2. authentication_controller.rb

There is also a tight integration between the controller name and its functions and the view folders and pages, which I'll elaborate on later. To begin with, simply create the two files above under the "controllers" folder:



And, for each controller, a stub class must be created:

authentication_controller.rb:

```
class AuthenticationController < ApplicationController
end
```

[Collapse](#) | [Copy Code](#)

home_controller.rb

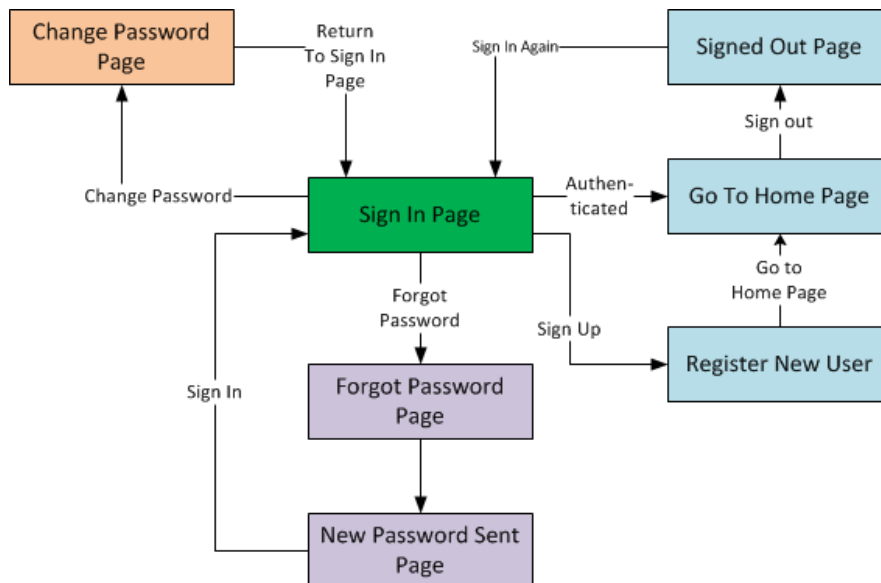
```
class HomeController < ApplicationController
end
```

[Collapse](#) | [Copy Code](#)

Note the Pascal-casing relationship between the filename and the class name. This is a requirement of Rails applications.

Views: The Authentication Pages

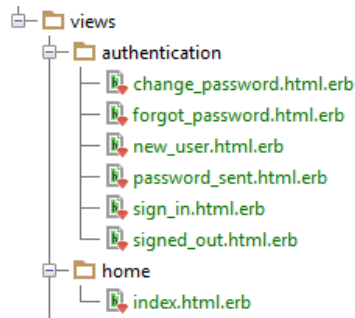
The following diagram illustrates the different pages we will want in our authentication system and the flow between them.



I always find it a question as to whether to start with the database model and backing entities or the views. If you are accustomed to a test driven environment, you may prefer to start with the tests. However, I'm going to leave testing to the second article.

In our particular case, except for the home page, we can implement the behavior of the authentication system inside one controller, supporting all six views (excluding the home page which we will assume is going to be handled by a separate view.) Also, only one model is required, the User model.

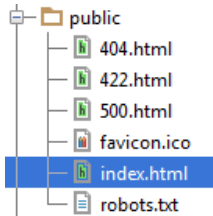
In the "views" folder, create a subfolder called "authentication" and in that folder, the six views. Also, create an index page for the home page in the subfolder (which you create) called "home". The result should look like this in the "views" folder:



Note how the subfolder name maps to the controller name!

Remove the Default Index Page

As the default Rails home page points out, you will need to delete the index.html file in the public folder:



Do so now.

Routes

Usually, each view requires a route entry in the routes.rb file.

The Root Route

Create your first route to the home page by adding to the routes.rb file (remember, this is in the config folder):

```
root :to=>"home#index"
```

[Collapse](#) | [Copy Code](#)

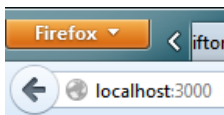
Note the above syntax, specifying "home" as the home controller and "index" as the page name!

If you put in some simple HTML to identify the home page (index.html.erb), for example:

```
<p>Home</p>
```

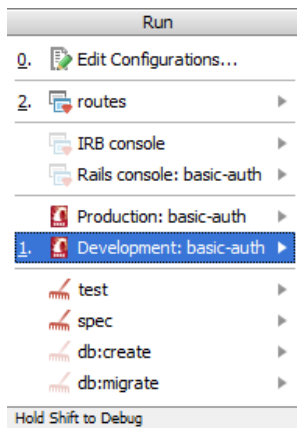
[Collapse](#) | [Copy Code](#)

you can then see that the new home page is rendered:



Home

Don't forget to start the server if it isn't already running -- because Ruby is interpreted, most of the time you will not need to restart the server to see changes. RubyMine will have the last command executed in the default "run" menu item, so you may need to explicitly specify that you want to run the web server. From the "Run" main menu, select "Run..." from the dropdown, then select "Development: basic-auth":



The Authentication Routes

Create the authentication routes for the six authentication pages:

[Collapse](#) | [Copy Code](#)

```
get "sign_in" => "authentication#sign_in"
get "signed_out" => "authentication#signed_out"
get "change_password" => "authentication#change_password"
get "forgot_password" => "authentication#forgot_password"
get "new_user" => "authentication#new_user"
get "password_sent" => "authentication#password_sent"
```

You can now visit these pages as well, and assuming you entered some simple HTML to render some text on each page, you will see each page:

```
localhost:3000/sign_in
localhost:3000/signed_out
localhost:3000/change_password
localhost:3000/forgot_password
localhost:3000/new_user
localhost:3000/password_sent
```

Important:

Note how the URL (such as "sign_in") directs Rails to the controller "AuthenticationController" and its method "sign_in" (if provided).

Also Important:

Each route will execute the function to the right of the '/' for the specified controller (the name to the left of the '/'). Rails is happy if the function is missing, as long as the controller is defined. If a function is implemented, it executes before the page is rendered, allowing you to initialize objects and set up other data that the page may require. The controller instance is available to the page, giving the page access to the controller's fields, which often includes the model with which the controller interfaces.

The Sign In Page: Controller, View, and Routes

To create a basic sign-in page, we need to work with the view itself, the controller, and the routes.rb files. The following renders as:

Sign In

Username or email:

Password:

Sign In

Clear Form

The View

Let's start with something basic:

[Collapse](#) | [Copy Code](#)

```
<p>Sign In</p>
<%= form_for @user, :as => :user, :url => sign_in_path(@user) do |f| %>
  <p>
```


Important:

1. the ":user" symbol as the backing model
2. the url matching a known route (see below)

Also note how we specify the text for the submit button "Sign In". If we didn't do this, Rails would default to the text "Create User", which is determined from the model specified in the "form_for" tag.

The Route

 Collapse | Copy Code

The Controller

[-] Collapse | Copy Code

Collapse | Copy Code

The second method handles the HTTP "post" command, determining whether the user is authenticated.

Error Messages

The code above needs to handle displaying authentication errors as well as preserving the current user during the session. Rails provides a mechanism for displaying messages to the user known as "flash." (read more [here](#))

application.html.erb

We can display messages created by a controller by adding a little bit of markup to the application.html.erb file, which as described above, is the global template for all pages, just before the yield call:

```
<% flash.each do |name, msg| %>
  <%= content_tag :div, msg, :id => "flash_#{name}" %>
<% end %>

<%= yield %>
```

[Collapse](#) | [Copy Code](#)

application.css

In the app\assets\stylesheets folder is the default css file. To make our alerts and notifications pretty, we can add some css:

```
#flash_error {
  border: 1px solid #000066;
  padding: 4px;
  margin-bottom: 10px;
  background-color: #cc3333;
}

#flash_notice {
  border: 1px solid #000066;
  padding: 4px;
  margin-bottom: 10px;
  background-color: #A3B15A;
}
```

[Collapse](#) | [Copy Code](#)

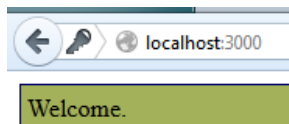
Controller Changes

and finally, we adjust the controller to utilize this new functionality:

```
if user
  flash[:notice] = 'Welcome.'
  redirect_to :root
else
  flash.now[:error] = 'Unknown user. Please check your username and password.'
  render :action => "sign_in"
end
```

[Collapse](#) | [Copy Code](#)

Now the user gets some notification of success:



Home

or failure:

Unknown user. Please check your username and password.

Sign In

Username or email:

marc.clifton@outlook.com

Password:

Sign In

Clear Form

Sessions

We also need to preserve the user in the session state, accomplished easily enough with:

```
if user
  session[:user_id] = user.id
  flash[:notice] = 'Welcome.'
  redirect_to :root
```

and we might also want to display the signed in username on all pages. In fact, we can now add "sign in", "sign out", and "sign up" links by checking the session state. First, we create a method that is accessible to the view, and this must be done in the application_controller.rb file, as each page renders going first through this controller:

[Collapse](#) | [Copy Code](#)

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

  def current_user
    # Note: we want to use "find_by_id" because it's OK to return a nil.
    # If we were to use User.find, it would throw an exception if the user can't be
    found.
    @current_user ||= User.find_by_id(session[:user_id]) if session[:user_id]
  end
end
```

Checking the user against the database, while a minor performance hit, ensures that if an administrator deletes a user, the next time the user navigates to a page, he/she will be automatically signed out. We will use this as part of the authentication tests later on.

And finally, we can create the links (globally for every page) depending on the login state (whether @current_user is nil or not), in the application.html.erb file, above the place where we display notifications:

[Collapse](#) | [Copy Code](#)

```
<p>
  <%= if current_user %>
    Logged in as <%= current_user.username %>.
    <%= link_to "Sign Out", signed_out_path %>
  <%= else %>
    <%= link_to "Sign up", new_user_path %> or
    <%= link_to "Sign In", sign_in_path %>
  <%= end %>
</p>
```

We now have a more functional home page:

Logged in as **cliftonm**. [Sign Out](#)

Welcome.

[Home](#)

The Signed Out Page

The signed out page is now simple to implement. We can add the method to the controller, display a notification, and clear the session's user id:

[Collapse](#) | [Copy Code](#)

```
def signed_out
  session[:user_id] = nil
  flash[:notice] = "You have been signed out."
end
```

[Sign up](#) or [Sign In](#)

You have been signed out.

Thank you for visiting.

The Sign Up Page

Sign Up

--

Clear Form

The Route

 Collapse | Copy Code

The Sign Up Page HTML

 Collapse | Copy Code

The show_field_error Function

Collapse | Copy Code

20/36

```

        </div>
      EOHTML
    end

    s.html_safe
  end
end

```

Here we have an interesting function which emits some HTML when an error exists, embedding the HTML similar to how data can be embedded into XML with the CDATA tag, but here we're using a token called "EOHTML."

The Authentication Controller

In the controller, we add two functions, one for instantiating a user field for the HTTP "get" command and the second function for handling the HTTP post command.

[Collapse](#) | [Copy Code](#)

```

def new_user
  @user = User.new
end

def register
  @user = User.new(params[:user])

  if @user.valid?
    @user.save
    session[:user_id] = @user.id
    flash[:notice] = 'Welcome.'
    redirect_to :root
  else
    render :action => "new_user"
  end
end

```

Notice how we initialize the User class, passing in an array of hashes (key-value pairs).

Notice how we are initializing a field (@user = User.new...) rather than a local variable (user = User.new...) in order to preserve whatever parameters the user had already entered if the form renders with errors.

A User Constructor

We therefore now need to specify a constructor for the User model:

[Collapse](#) | [Copy Code](#)

```

class User < ActiveRecord::Base
  # ...
  def initialize(attributes = {})
    super # must allow the active record to initialize!
    attributes.each do |name, value|
      send("#{name}=", value)
    end
  end
  # ...
end

```

For each key-value pair (hash) we assign the value to the attribute by calling the "send" function (all method calls in Ruby are actually messages.)

Important:

We don't actually need to do this for the User class because the constructor provided by Rails will allow us to do a "mass assign" from a hash as long as the fields that we are assigning have been designated as "attr_accessible", which they have. However, there are cases when one wants to initialize several fields (such as ID's in a many-to-many table) that are not intended to be accessible to a view but instead are designated with an "attr_accessor" instead. The above function is a simple way of providing safe mass assign capability for internal constructors.

Testing The Validations

One of the simplest ways to test the validations is to try to Sign Up with a blank form, which results in this rendering:

[Sign up](#) or [Sign In](#)

Sign Up

Username:

can't be blank

Email:

can't be blank

Password:

can't be blank

Password confirmation:

We can also test that duplicate usernames and email addresses are validated, as well as that the password and password confirmation match.

The Change Password Page (Settings)

Logged in as cliftonm. [Sign Out](#) [Account Settings](#)

Account Settings

Current password:

Username: (leave blank if you are not changing your username)

Email: (leave blank if you are not changing your email address)

New password: (leave blank if you are not changing your password)

Password confirmation:

This page is very similar to the Sign Up page, however it doesn't require a username or email address, but does require the user to enter his/her existing password. As I write this, it occurs to me that a more useful implementation would be to allow the user to change their username, email address, and/or password in an 'account settings' page, so we'll implement that.

An Account Settings Link

First, we'll add a link for "Account Settings" that the user can access when logged in, which goes into the application.html.erb "common to all pages" template:

[Collapse](#) | [Copy Code](#)

```
<% if current_user %>
  Logged in as <%= current_user.username %>.
  <%= link_to "Sign Out", signed_out_path %>
  <%= link_to "Account Settings", account_settings_path %>
<% end %>
```

We'll need a couple routes to handle the HTTP "get" and "post" commands:

[Collapse](#) | [Copy Code](#)

```
get "account_settings" => "authentication#account_settings"
put "account_settings" => "authentication#set_account_info"
```

Because the user record is already populated when Rails processes the "form_for" tag, it issues an HTTP "put" command to update the existing record, hence the above route uses "put" instead of "post." Oddly enough though, if errors occur on the page the Update button is clicked again, Rails generates a "post" command, therefore, in the "form_for" tag, I explicitly define the HTTP command to be put, with the optional ":method => :put".

The User Model

The new password and confirmation fields need to be added to the user model:

[Collapse](#) | [Copy Code](#)

```
attr_accessible :email, :username, :password, :password_confirmation,
:new_password, :new_password_confirmation
attr_accessor :password, :new_password
```

The above ensures that the attributes (what I've been calling "fields") are accessible to the view as well as being backed by actual fields in the model, even though they do not exist in the User database table.

We also want to do confirmation of a new password only if the new password field is not blank (otherwise this messes up the sign up page):

[Collapse](#) | [Copy Code](#)

```
validates_confirmation_of :new_password, :if => Proc.new {|user|
!user.new_password.nil? && !user.new_password.empty? }
```

This validation runs only when the new_password field is not nil and not empty.

We have to a same thing with the email and username "presence_of" and "uniqueness_of" validations so they work correctly for both the sign in and account settings pages:

[Collapse](#) | [Copy Code](#)

```
validates_presence_of :email, :if => Proc.new {|user|
  user.previous_email.nil? || user.email != user.previous_email}

validates_presence_of :username, :if => Proc.new {|user|
  user.previous_username.nil? || user.username != user.previous_username}

validates_uniqueness_of :email, :if => Proc.new {|user|
  user.previous_email.nil? || user.email != user.previous_email}

validates_uniqueness_of :username, :if => Proc.new {|user|
  user.previous_username.nil? || user.username != user.previous_username}
```

The Authentication Controller

For the HTTP "get" command, we simply assign the currently signed in user record (using the method we've already created) to the field:

[Collapse](#) | [Copy Code](#)

```
def account_settings
  @user = current_user
end
```

This very simply populates the form with the existing fields. In the screenshot above, the password field is also populated, this is an artifact of the browser having a cookie for my password -- since the backing database table doesn't have a password field, the browser is not getting this value from our application. This is easily enough confirmed by setting a breakpoint on the assignment and inspecting the record being returned by the current_user function call.

The code for the HTTP "put" command is a little more complicated because we have to update certain fields depending on what the user changed. Furthermore, we have to confirm the current user's password.

[Collapse](#) | [Copy Code](#)

```
def set_account_info
  old_user = current_user

  # verify the current password by creating a new user record.
  @user = User.authenticate_by_username(old_user.username, params[:user]
[:password])

  # verify
  if @user.nil?
    @user = current_user
    @user.errors[:password] = "Password is incorrect."
    render :action => "account_settings"
  else
    # update the user with any new username and email
    @user.update(params[:user])
    # Set the old email and username, which is validated only if it has changed.
    @user.previous_email = old_user.email
    @user.previous_username = old_user.username

    if @user.valid?
      # If there is a new_password value, then we need to update the password.
```

```

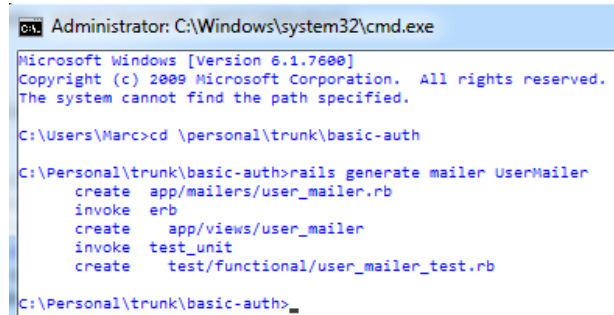
    @user.password = @user.new_password unless @user.new_password.nil? ||
@user.new_password.empty?
    @user.save
    flash[:notice] = 'Account settings have been changed.'
    redirect_to :root
  else
    render :action => "account_settings"
  end
end
end
end

```

We now have a flexible solution for the user to administrate their account settings.

Sending Mail

Before proceeding with the last screen, "forgot password", we need to set up the ability to mail instructions for resetting the password. This involves setting up the Rails ActionMailer, as described [here](#). Open a console window (Start -> "cmd") and change the directory to the application folder, then run the command "rails generate mailer UserMailer":



```

Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
The system cannot find the path specified.

C:\Users\Marc>cd \personal\trunk\basic-auth

C:\Personal\trunk\basic-auth>rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create  app/views/user_mailer
invoke  test_unit
create  test/functional/user_mailer_test.rb

C:\Personal\trunk\basic-auth>_

```

Adding a Welcome Email

Let's try this first with a "welcome" email when the user first signs up. As per the documentation, create a welcome_email method:

[Collapse](#) | [Copy Code](#)

```

def welcome_email(user)
  @user = user
  @url = "<a
href='http://localhost:3000/sign_in'>http://localhost:3000/sign_in</a>"
  @site_name = "localhost"
  mail(:to => user.email, :subject => "Welcome to my website.")
end

```

and as the documentation suggests, create an HTML and text templates (note that the base filename for these two files matches the function name!). In app/views/user_mailer, create welcome_email.html.erb:

[Collapse](#) | [Copy Code](#)

```

<!DOCTYPE html>
<html>
<head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
</head>
<body>
<h2>Welcome to <%= @site_name %>, <%= @user.username %></h2>
<p>
You have successfully signed up to <%= @site_name %>,
your username is: <%= @user.username %>.<br/>
</p>
<p>
To login to the site, just follow this link: <%= @url %>.
</p>
<p>Thanks for joining and have a great day!</p>
</body>
</html>

```

and also welcome_email.text.erb:

[Collapse](#) | [Copy Code](#)

```

Welcome to <%= @site_name %>, <%= @user.username %>
=====

You have successfully signed up to <%= @site_name %>,
your username is: <%= @user.username %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!

```


Updating the Authentication Controller

In the authentication_controller.rb file, add a call to "register" function for to send the welcome email:

[Collapse](#) | [Copy Code](#)

```
def register
  @user = User.new(params[:user])

  if @user.valid?
    @user.save
    UserMailer.welcome_email(@user).deliver
    session[:user_id] = @user.id
    flash[:notice] = 'Welcome.'
    redirect_to :root
    # ...
  end
end
```

Configure the Mailer

Configure the mailer to send the email via your mail server. For this example, we'll use Gmail (assuming you have an account on Gmail). Open the config/environment/development.rb file (since we're in development mode) and add:

[Collapse](#) | [Copy Code](#)

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  :address => "smtp.gmail.com",
  :port => 587,
  :domain => '<your domain>',
  :user_name => '<your username>',
  :password => '<your password>',
  :authentication => 'plain',
  :enable_starttls_auto => true }
```

If all goes well, you should receive an email that look similar to this:

Welcome to localhost, cliftonm

You have successfully signed up to localhost, your username is: cliftonm.

To login to the site, just follow this link: http://localhost:3000/sign_in.

Thanks for joining and have a great day!

Using Environment Variables

Rather than hardcoding your username and password into the development.rb environment file, it's a much better idea to use environment variables - for example, given that I am pushing this project onto a public GitHub repository, I don't want people to see my Gmail username and password! There are several options that are available as documented [here](#). The approach I'm taking is to create a file that is specifically ignored by Git:

Create a local_env.yml file in the config folder with the contents:

[Collapse](#) | [Copy Code](#)

```
GMAIL_USERNAME: '<your username>'
GMAIL_PASSWORD: '<your password>'
```

Obviously replacing the username and password with your own.

Edit the .gitignore file (at the root of your application folder) and add:

[Collapse](#) | [Copy Code](#)

```
/config/local_env.yml
```

This ensures that Git will not commit or push this file to the remote repository.

Edit the application.rb file, found in the config folder, adding:

[Collapse](#) | [Copy Code](#)

```
config.before_configuration do
  env_file = File.join(Rails.root, 'config', 'local_env.yml')
  YAML.load(File.open(env_file)).each do |key, value|
    ENV[key.to_s] = value
  end if File.exists?(env_file)
end
```

just after the line "config.assets.version = '1.0'".

Restart the server and test your email again!

Forgetting Your Password

Now that we have email working, we can add the final web page that provides the user with instructions on how to reset their password. We don't want to change their password in case someone else is trying to get access to their account, instead, we only send an email with instructions on how to reset the password. The email address must already be a registered user, and the URL provided must include a unique token that we also store in the database and must match when the user resets their password. There is an [excellent Railscast](#) on how to this, which I am borrowing from.

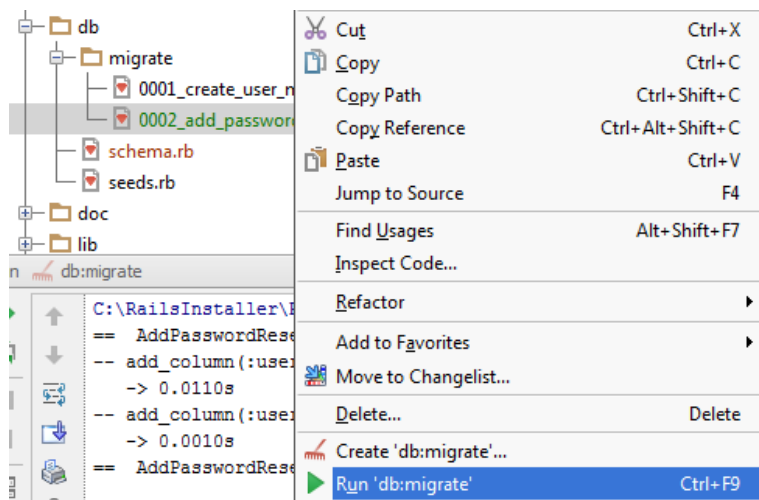
First, we'll use a database migration to add a password reset token field and a date field so that the reset opportunity expires within 24 hours. We'll call the migration file "0002_add_password_reset_fields.rb", and the contents will be:

```
class AddPasswordResetFields < ActiveRecord::Migration
  def self.up
    add_column :users, :password_reset_token, :string
    add_column :users, :password_expires_after, :datetime
  end

  def self.down
    remove_column :users, :password_reset_token
    remove_column :users, :password_expires_after
  end
end
```

[Collapse](#) | [Copy Code](#)

Run the migration by right-clicking on the migration folder and selecting "db:migrate" from the RubyMine folder tree:



The Forgot My Password View

[Sign up](#) or [Sign In](#)

Forgot Password

Username or email:

We'll use the User model for this form as well, but only request that the user enters in their username or email address, which we'll then verify in the database as an active user:

```
<p>Forgot Password</p>
<%= form_for @user, :as => :user, :url => forgot_password_path, :method => :put do
|f| %>
  <p>
    <%= f.label 'username or email:' %><br/>
    <%= f.text_field :username %>
    <%= show_field_error(@user, :username) %>
  </p>
  <p>
    <%= f.submit 'Send Password Reset Instructions' %>
  </p>
<% end %>
```

[Collapse](#) | [Copy Code](#)

The Password Reset View

[Sign up](#) or [Sign In](#)

Account Settings

cliftonm has requested a password reset. Please enter your new password:

New password:

••••••••

Password confirmation:

Update

We also need a view to handle the password reset:

[Collapse](#) | [Copy Code](#)

```
<p>Account Settings</p>
<%= form_for @user, :as => :user, :url => password_reset_path, :method => :put do
|f| %>
  <p>
    <%= @user.username %> has requested a password reset. Please enter your new
password:
  </p>
  <p>
    <%= f.label 'new password:' %><br/>
    <%= f.password_field :new_password %>
    <%= show_field_error(@user, :new_password) %>
  </p>
  <p>
    <%= f.hidden_field :username %>
    <%= f.label 'password confirmation:' %><br/>
    <%= f.password_field :new_password_confirmation %>
    <%= show_field_error(@user, :new_password_confirmation) %>
  </p>
  <p>
    <%= f.submit 'Update' %>
  </p>
<% end %>
```

Note how we are using the hidden field "username" so that on the HTTP put command, we can re-acquire the username (which we know is unique) and thus the user record on which to update the password.

A "Forgot Your Password" Link

[Sign up](#) or [Sign In](#)

Sign In

Username or email:

marc.clifton@gmail.com

Password:

••••••••

Sign In

Clear Form

[forgot your password?](#)

We also need a link on the sign in page for the user to click:

[Collapse](#) | [Copy Code](#)

```
<p>
  <%= link_to 'forgot your password?', :forgot_password %>
</p>
```

Routes

As usual, we need some routes:

[Collapse](#) | [Copy Code](#)

```
get "forgot_password" => "authentication#forgot_password"
put "forgot_password" => "authentication#send_password_reset_instructions"

get "password_reset" => "authentication#password_reset"
put "password_reset" => "authentication#new_password"
```

The first pair handles the page on which the user requests a password reset. The second pair

handles the page that allows the user to enter a new password.

The Password Reset Email

The UserMailer class gets a new function:

[Collapse](#) | [Copy Code](#)

```
def reset_password_email(user)
  @user = user
  @password_reset_url = 'http://localhost:3000/password_reset?' +
    @user.password_reset_token
  mail(:to => user.email, :subject => 'Password Reset Instructions.')
end
```

Obviously there are some hardcoded values that we would want to replace for a production deployment, preferably using environment variables or some other approach.

Backed by the email "view" reset_password_email.html.erb:

[Collapse](#) | [Copy Code](#)

```
<!DOCTYPE html>
<html>
<head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
</head>
<body>
<p>
  <%= @user.username %> has requested a password reset.
</p>
<p>
  To reset your password, click the URL below.
</p>
<p>
  <%= @password_reset_url %>
</p>
<p>
  If you did not request your password to be reset, just ignore this email and your
  password will continue to stay the same.
</p>
</body>
</html>
```

The Authentication Controller

The controller needs functions for the HTTP get and put commands for both requesting a password reset and resetting the password.

Requesting a Password Reset

[Collapse](#) | [Copy Code](#)

```
# HTTP get
def forgot_password
  @user = User.new
end

# HTTP put
def send_password_reset_instructions
  username_or_email = params[:user][:username]

  if username_or_email.rindex('@')
    user = User.find_by_email(username_or_email)
  else
    user = User.find_by_username(username_or_email)
  end

  if user
    user.password_reset_token = SecureRandom.urlsafe_base64
    user.password_expires_after = 24.hours.from_now
    user.save
    UserMailer.reset_password_email(user).deliver
    flash[:notice] = 'Password instructions have been mailed to you. Please check
your inbox.'
    redirect_to :sign_in
  else
    @user = User.new
    # put the previous value back.
    @user.username = params[:user][:username]
    @user.errors[:username] = 'is not a registered user.'
    render :action => "forgot_password"
  end
end
```

The more interesting part of the code above is the function that handles the HTTP put command. Here, we verify that the username or email address matches a user record in the database, and if so, we create an expiration token and set the expiration date for 24 hours later, then deliver the notification email. The email the user receives looks like this:

cliftonm has requested a password reset.

To reset your password, click the URL below.

http://localhost:3000/password_reset?lnY-nqObK4Azsw9kr91mw

If you did not request your password to be reset, just ignore this email and your password will continue to stay the same.

Resetting the Password

Once the user clicks on the link provided by the email, we need to process validating the reset. When the user first lands on the page, we verify that the token is associated with a user and that the time to reset the password has not expired:

[Collapse](#) | [Copy Code](#)

```
# The user has landed on the password reset page, they need to enter a new
password.
# HTTP get
def password_reset
  token = params.first[0]
  @user = User.find_by_password_reset_token(token)

  if @user.nil?
    flash[:error] = 'You have not requested a password reset.'
    redirect_to :root
    return
  end

  if @user.password_expires_after < DateTime.now
    clear_password_reset(@user)
    @user.save
    flash[:error] = 'Password reset has expired. Please request a new password
reset.'
    redirect_to :forgot_password
  end
end
```

When the user changes his/her password, we verify that the password is not blank and that it matches the confirmation. We need to do this manually because the user model validation routine will not validate a blank "new_password", as this is desired behavior for the account administration page. So we add the custom validation here.

[Collapse](#) | [Copy Code](#)

```
# The user has entered a new password. Need to verify and save.
# HTTP put
def new_password
  username = params[:user][:username]
  @user = User.find_by_username(username)

  if verify_new_password(params[:user])
    @user.update(params[:user])
    @user.password = @user.new_password

    if @user.valid?
      clear_password_reset(@user)
      @user.save
      flash[:notice] = 'Your password has been reset. Please sign in with your new
password.'
      redirect_to :sign_in
    else
      render :action => "password_reset"
    end
  else
    @user.errors[:new_password] = 'Cannot be blank and must match the password
verification.'
    render :action => "password_reset"
  end
end
```

And we have a couple helper functions:

[Collapse](#) | [Copy Code](#)

```
private

def clear_password_reset(user)
  user.password_expires_after = nil
  user.password_reset_token = nil
end

def verify_new_password(passwords)
  result = true

  if passwords[:new_password].blank? || (passwords[:new_password] !=
passwords[:new_password_confirmation])
    result = false
  end

  result
end
```

Important:

I encounter a lot of Ruby/Rails code in which functionality is duplicated (copy and paste is dangerously easy in Ruby) and functions containing blocks of code that, in addition to being undocumented, are also doing many different things. It would be much clearer to a developer coming into a project if the code was cleaner, and by this I mean, avoiding duplicate code and breaking apart a function into several smaller functions. Even the code that I'm presenting in this article could be improved in that regard. Another thing about Ruby on Rails is that you usually don't need to write a lot of code to do one thing - typically, that thing can be done in a line or two. This means that functions are a very tight composition of "do A, B, C, D, E", probably taking up no more than five lines. A perfect example is something like this:

[Collapse](#) | [Copy Code](#)

```
1: user.password_reset_token = SecureRandom.urlsafe_base64
2: user.password_expires_after = 24.hours.from_now
3: user.save
4: UserMailer.reset_password_email(user).deliver
5: flash[:notice] = 'Password instructions have been mailed to you. Please check
  your inbox.'
6: redirect_to :sign_in
```

And as you can see, there's a lot going on here:

line 1 & 2 is initializing the fields responsible for managing a password reset
 line 3 is updating the database
 line 4 is sending an email
 line 5 is setting up to display a notice on the screen
 line 6 redirects the browser to the sign in page

Basically, we're talking to five separate "entities" here:

1. the model
2. the ORM
3. the email subsystem
4. the notification subsystem
5. the page management subsystem

That's a lot of mental context switching, especially for a new developer. Code clarity (small functions and comments) is important to maintain the legibility and maintainability of Ruby on Rails code.

Remember Me

Let's add a "remember me" checkbox to the sign in and sign up pages. We will add an "authentication_token" to the User table by creating a new migration
 0003_add_authentication_token.rb:

[Collapse](#) | [Copy Code](#)

```
class AddAuthenticationToken < ActiveRecord::Migration
  def self.up
    add_column :users, :authentication_token, :string
  end

  def self.down
    remove_column :users, :authentication_token
  end
end
```

Next, we add a "remember_me" attribute to the user model:

[Collapse](#) | [Copy Code](#)

```
attr_accessible :remember_me, [etc...]
attr_accessor :remember_me, [etc...]
```

Next, we add the checkbox to the sign in and sign up pages:

[Collapse](#) | [Copy Code](#)

```
<p>
  <%= f.check_box :remember_me %>
</p>
```

Now we can modify our controller to generate the authentication token and save it both as a cookie and to the database if the checkbox is checked. If the checkbox is unchecked or the user signs out, we clear this token. We modify the sign in controller function:

[Collapse](#) | [Copy Code](#)

```
def login
  username_or_email = params[:user][:username]
  user = verify_user(username_or_email)

  if user
    update_authentication_token(user, params[:user][:remember_me])
    user.save
  end
end
```

The register user is similar:

[Collapse](#) | [Copy Code](#)

```
if @user.valid?
  update_authentication_token(@user, nil)
  @user.save
```

The sign out clears the authentication token:

[Collapse](#) | [Copy Code](#)

```
def signed_out
  # clear the authentication token when the user manually signs out
  user = User.find_by_id(session[:user_id])

  if user
    update_authentication_token(user, nil)
    user.save
    session[:user_id] = nil
    flash[:notice] = "You have been signed out."
  else
    redirect_to :sign_in
  end
end
```

and lastly, we have the common method being called here:

[Collapse](#) | [Copy Code](#)

```
def update_authentication_token(user, remember_me)
  if remember_me == 1
    # create an authentication token if the user has clicked on remember me
    auth_token = SecureRandom.urlsafe_base64
    user.authentication_token = auth_token
    cookies.permanent[:auth_token] = auth_token
  else # nil or 0
    # if not, clear the token, as the user doesn't want to be remembered.
    user.authentication_token = nil
    cookies.permanent[:auth_token] = nil
  end
end
```

Now all that's left is to acquire the current user from the cookie in the application controller's `current_user` method:

[Collapse](#) | [Copy Code](#)

```
def current_user
  # Note: we want to use "find_by_id" because it's OK to return a nil.
  # If we were to use User.find, it would throw an exception if the user can't be
  found.
  @current_user ||= User.find_by_id(session[:user_id]) if session[:user_id]
  @current_user ||= User.find_by_authentication_token(cookies[:auth_token]) if
  cookies[:auth_token] && @current_user.nil?
  @current_user
end
```

This is a little more complicated than it necessary, as we could use a non-permanent cookie to also preserve the authentication token and thus wouldn't need the session user ID, however, I prefer to leave the existing code in place as it makes it more obvious that we are trying to acquire the current user from either the session or the cookie.

Using Recaptcha

Lastly, we want some protection from malfeasants registering into our application, so we'll add a Recaptcha field to the Sign Up page. We will use [this recaptcha gem](#). First, add to your gemfile:

[Collapse](#) | [Copy Code](#)

```
gem 'recaptcha', :require => "recaptcha/rails"
```

Then, in RubyMine, run the Bundler -> Install from the Tools menu.

Create your keys from the [Google recaptcha website](#).

Create a file called `recaptcha.rb` in the `config/initializers` folder, with the contents similar to:

[Collapse](#) | [Copy Code](#)

```
Recaptcha.configure do |config|
  config.public_key = '6Lc6BAAAAAACHqRbQZcn_yyyyyyyyyyyyyyyy'
  config.private_key = '6Lc6BAAAAAANK3DRm6VA_xxxxxxxxxxxxxxxxxx'
end
```

Then, in the `new_user.html.erb` view, we add `"recaptcha_tags"` before the sign up button:

[Collapse](#) | [Copy Code](#)

```
<p>
<%= recaptcha_tags %>
</p>
<p>
```

After restarting the server, we now get a repatcha block in our new user registration page:

Now we need to modify the register method in our authentication controller to verify the recaptcha text and flash an appropriate message if it is incorrect:

 Collapse | Copy Code

And we're done adding the recaptcha feature!

We can use our authentication system to authenticate our own web pages! In our particular case, the account settings page (both HTTP post and put commands) should not be accessible to the user when not logged in:

[-] Collapse | Copy Code

This tells Rails to execute the method "authenticate_user" for the "account_settings" and "set_account_info" functions.

Important:

The "authenticate_user" method must be implemented in the application_controller.rb, as all our controllers are derived from ApplicationController (it's important to understand the difference between application_controller.rb and application_helper.rb [see above]):

[Collapse](#) | [Copy Code](#)

```
def authenticate_user
  if current_user.nil?
    flash[:error] = 'You must be signed in to view that page.'
    redirect_to :root
  end
end
```

Now, when the page is accessed without being signed in, the user is presented with:

[Sign up](#) or [Sign In](#)

You must be signed in to view that page.

[Home](#)

A Final Touch

Signed Up and Last Signed In DateTime

There's a few bells and whistles we could add, such as "signed up datetime" and "last signed in datetime", which is very simple simple - alter the user table with a database migration...

[Collapse](#) | [Copy Code](#)

```
class AddDateInfoFields < ActiveRecord::Migration
  def self.up
    add_column :users, :signed_up_on, :datetime
    add_column :users, :last_signed_in_on, :datetime
  end

  def self.down
    remove_column :users, :signed_up_on
    remove_column :users, :last_signed_in_on
  end
end
```

...then adding the appropriate fields and then adding a couple lines of code in the controller to set the values. In the login function, before the call to "user.save":

[Collapse](#) | [Copy Code](#)

```
user.last_signed_in_on=DateTime.now
```

and in the register function, before the call to "@user.save":

[Collapse](#) | [Copy Code](#)

```
@user.signed_up_on = DateTime.now
@user.last_signed_in_on = @user.signed_up_on
```

Adminstrating Users

Logged in as cliftonm. [Sign Out](#) [Account Settings](#) [Admin](#)

Admin Users

Username	Signed Up On	Last Signed In On	
cliftonm	2013-04-10 14:15:19 UTC	2013-04-10 14:15:19 UTC	Delete

Obviously this (and all the other pages here) could be made a lot more sexier with stylesheets and other useful functions, but that isn't the main point of this article.

We'll add a simple administration screen that allows us to delete users. Obviously this screen should only be available to administrators, but for now we're going to ignore authorization (which, by the way, [CanCan](#) is a very popular gem to use and integrates well with [Devise](#).)

First, the view, which I created in the "views" folder, under a new folder called "admin", as "users.html.erb":

[Collapse](#) | [Copy Code](#)

```
<p>Admin Users</p>
<table>
  <tr>
```

```

<th width="30%">Username</th>
<th width="20%">Signed Up On</th>
<th width="20%">Last Signed In On</th>
</tr>
<% for user in @users %>
  <tr>
    <td><%= user.username %></td>
    <td><%= user.signed_up_on %></td>
    <td><%= user.last_signed_in_on %></td>
    <td><%= link_to 'Delete', user, :confirm => 'Are you sure?', :method =>
      :delete %></td>
  </tr>
<% end %>
</table>

```

Then we need a couple routes:

[Collapse](#) | [Copy Code](#)

```

get "admin_users" => "admin#users"
delete "user/:id" => "admin#delete_user", :as => "user"

```

and finally, the new admin_controller.rb file:

[Collapse](#) | [Copy Code](#)

```

class AdminController < ApplicationController
  def users
    @users = User.all
  end

  def delete_user
    if params[:id] == current_user.id.to_s
      flash.now[:error] = 'You cannot delete yourself!'
      @users = User.all
      render :action => :users
    else
      User.find_by_id(params[:id]).delete
      @users = User.all
      render :action => :users
    end
  end
end

```

The first function "users" is called from the HTTP "get" command and simply loads all the users into an attribute (field) which is accessible to the view. The view iterates over the collection, creating rows for each record. The "delete" link is given the user record instance, and Rails automatically sends this link as, for example: "<http://localhost:3000/user/20>". The "delete" route handles the HTTP "delete" command, calling the delete_user function in the admin controller. Here, we first verify that the user being deleted is not oneself, which we disallow, otherwise, we proceed, deleting the user, refreshing the list and re-rendering the same page.

Source Code

The source code is available on GitHub: <https://github.com/cliftonm/basic-auth>

What's Next?

Testing is a vital part of developing websites with Ruby on Rails. In the next installment, we'll write some test scenarios for our basic authentication system. After that, I will be looking at implementing a role authorization component to complete the authentication / authorization tutorials. Once these are in place, I'll write a tutorial on how to put a forum together in Rails -- nothing as sophisticated as Code Project's forum, but considerably more so than the forum gems that I've tried out that are available for Rails at the time of this writing. This may be followed up by an idea that I have for a community-based website.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author

Marc Clifton

United States 



Marc is the creator of two open source projects, [MyXaml](#), a declarative (XML) instantiation engine and [the Advanced Unit Testing framework](#), and [Interacx](#), a commercial n-tier RAD application suite. Visit his website, www.marcclifton.com, where you will find many of his articles and his [blog](#).

Marc lives in Philmont, NY.

[Article Top](#)

Comments and Discussions

You must [Sign In](#) to use this message board.

Search this forum

Go

☒ Profile popups Spacing

Relaxed

 Noise

Medium

 Layout




























Thread View

 Per page

50

Update


First Prev Next


 Thanks, very useful! 	 Member 10994024	17hrs 17mins ago	1
 Thank you 	 Member 10839370	23-May-14 2:22	1
 My vote of 5 	 hscott123	3-Sep-13 16:30	2
 My vote of 5 	 DrABELL	12-Aug-13 7:30	1
 My vote of 5 	 L Hills	22-Apr-13 5:52	1
 My vote of 5 	 linuxjr	10-Apr-13 6:58	2
 My vote of 5 	 Pete O'Hanlon	10-Apr-13 6:38	3
 My vote of 1 	 Rage	10-Apr-13 6:00	2
 My vote of 5 	 <i>Orcun lyigun</i>	10-Apr-13 4:03	2


Last Visit: 31-Dec-99 18:00 Last Update: 6-Aug-14 3:29


Refresh


1


 General


 News


 Suggestion


 Question

 Bug

 Answer

 Joke

 Rant



Admin

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Mobile](#)
Web01 | 2.8.140721.1 | Last Updated 10 Apr 2013

Layout: [fixed](#) | [fluid](#)

Article Copyright 2013 by Marc Clifton
Everything else Copyright © [CodeProject](#), 1999-2014
[Terms of Service](#)