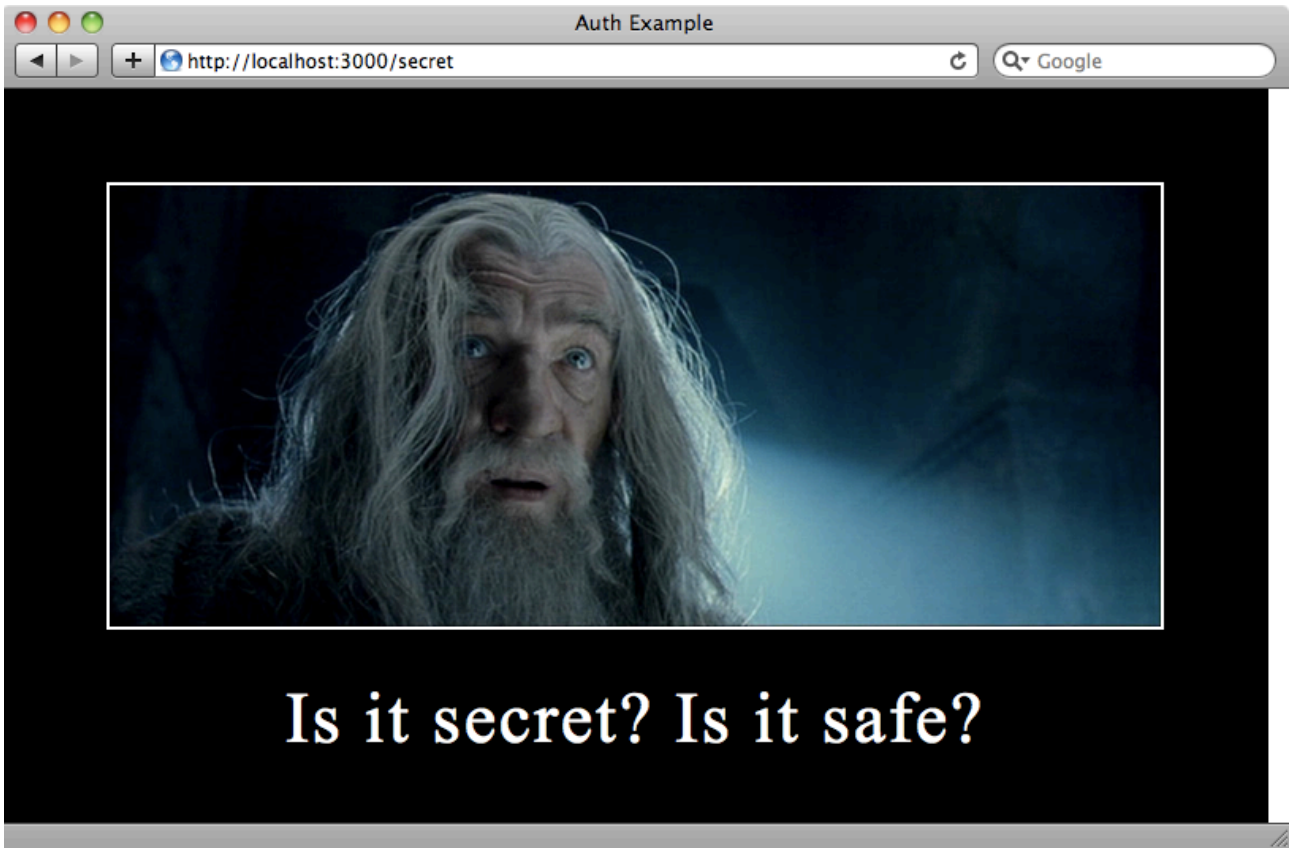
Episode 270

Authentication in Rails 3.1

Rails 3.1 come with new features related to authentication and in this episode we'll show you how some of them work. To demonstrate them we have a simple application with a page that can be accessed by anyone who knows the URL.



HTTP Basic Authentication

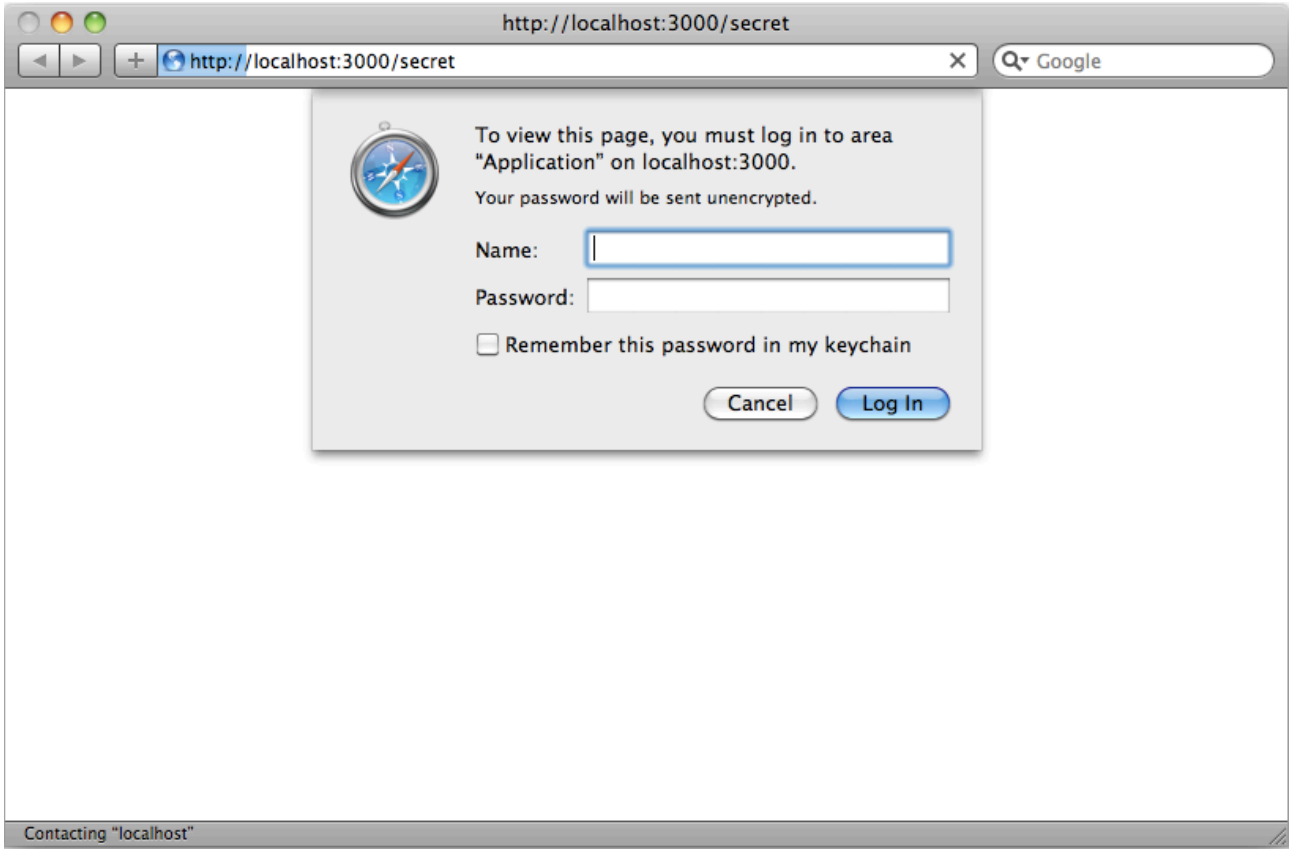
We want to add authorization to this page so that it can only be accessed by certain users. For this we need to add some authentication and the quickest way to add this is to add HTTP Basic authentication. Rails 3.1 gives us a new way to do this; all we need to do is to modify the page's controller and add a call to a new method called `http_basic_authenticate_with` which takes :`name` and :`password` options.

/app/controllers/secret_controller.rb

```
class SecretController < ApplicationController
  http_basic_authenticate_with :name => "frodo", ↵
    :password => "thering"
  def index
  end
end
```

If we want to restrict the authentication to certain actions we can add an `:only` or `:except` option. Obviously in a real application we'd move the username and password into a config file rather than having it in plain view in the code but for this demo application it can be left as it is.

When we visit the page now we'll see a login dialogue and we'll only be able to view the page by entering the correct username and password we specified in `http_basic_authenticate_with`.



HTTP Basic is a little under-utilized. It's quick and easy to add whenever we need to make part of a site private and in Rails 3.1 it's even easier to implement.

Using secure_password

Sometimes though we need a more comprehensive authorization system that can handle multiple users and to help with that Rails 3.1 gives us `secure_password`.

Back in episode 250 [watch[1], read[2]] we created an authentication system from scratch and Rails 3.1 makes doing this quite a bit easier. We'll do this now and replace the HTTP Basic authentication with our own.

The first thing to do is to generate a `User` model with fields for an email address and a password.

```
$ rails g model user email:string password_digest:string
```

We can then create the database table by migrating the database.

```
$ rake db:migrate
```

The important thing here is that we call the field that will store passwords `password_digest`. Next we'll need to add a call to `has_secure_password` in the `User` model.

/app/models/user.rb

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

This adds methods to set and authenticate the entered password, adds validators for the password and password confirmation and also adds authentication functionality. The `password_digest` field we created earlier is used in the background to store the hashed password.

By default there's no `validates_presence_of` validator for the password so we'll add one that fires when a new user is created.

/app/models/user.rb

```
class User < ActiveRecord::Base
  has_secure_password
  validates_presence_of :password, :on => :create
end
```

---

[1] http://railscasts.com/episodes/250-authentication-from-scratch

[2] http://asciicasts.com/episodes/250-authentication-from-scratch

Normally we'd want to validate the email address too but we won't to that here.

To allow users to create accounts we'll create a `UsersController`.

```
$ rails g controller users
```

The code for the controller is fairly standard and looks like this:

/app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to root_url, :notice => "Signed up!"
    else
      render "new"
    end
  end
end
```

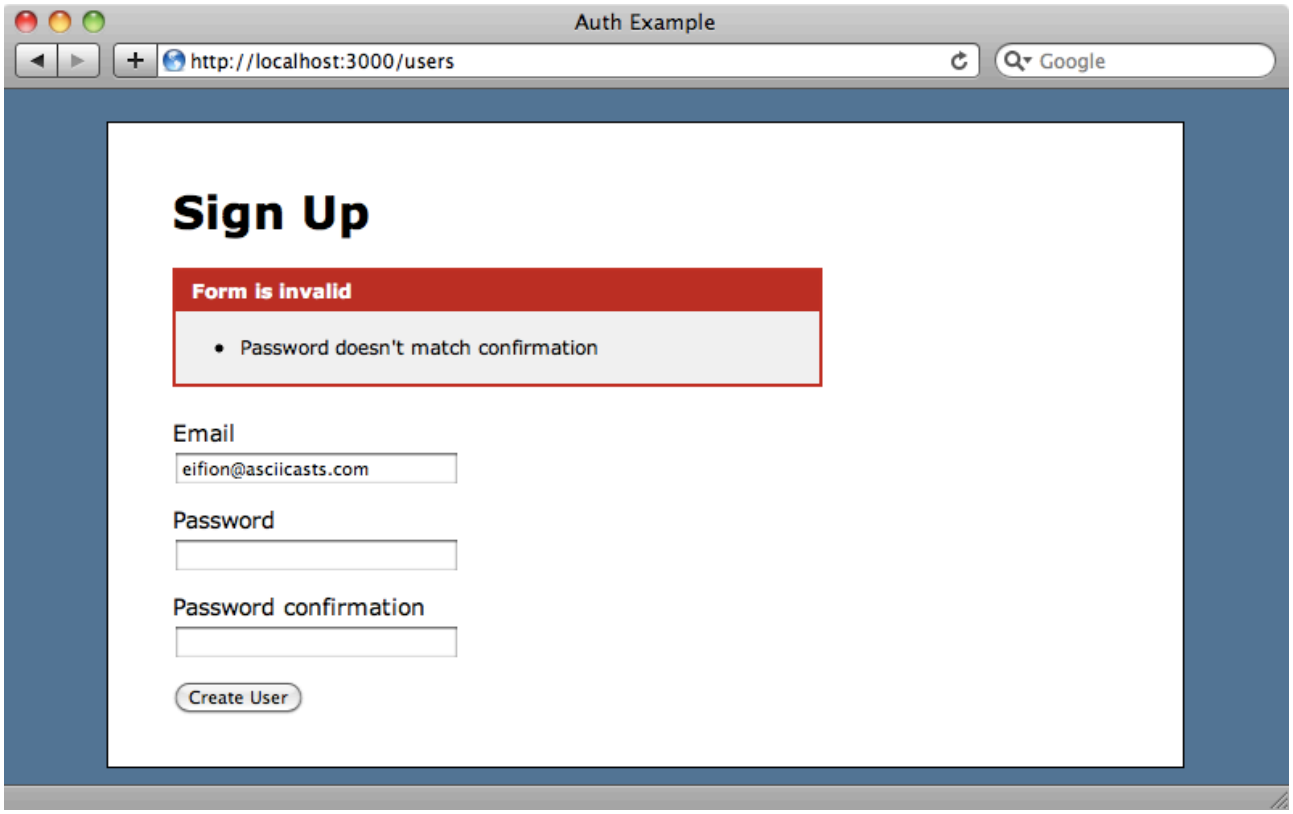
The new view contains a form for signing up.

/app/views/users/new.html.erb

```
<h1>Sign Up</h1>

<%= form_for @user do |f| %>
  <% if @user.errors.any? %>
    <div class="error_messages">
      <h2>Form is invalid</h2>
      <ul>
        <% for message in @user.errors.full_messages %>
          <li><%= message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <div class="field">
    <%= f.label :email %>
    <%= f.text_field :email %>
  </div>
  <div class="field">
    <%= f.label :password %>
    <%= f.password_field :password %>
  </div>
  <div class="field">
    <%= f.label :password_confirmation %>
    <%= f.password_field :password_confirmation %>
  </div>
  <div class="actions"><%= f.submit %></div>
<% end %>
```

If we try signing up and enter a password that doesn't match the confirmation we'll see the validation message that was provided by `has_secure_password`.



When we enter a matching password confirmation we'll be signed up successfully.

Users can sign up but not yet sign in so next we'll create a login form with `email` and `password` fields.

/app/views/sessions/new.html.erb

```
<h1>Log in</h1>

<%= form_tag sessions_path do %>
  <div class="field">
    <%= label_tag :email %>
    <%= text_field_tag :email, params[:email] %>
  </div>
  <div class="field">
    <%= label_tag :password %>
    <%= password_field_tag :password %>
  </div>
  <div class="actions"><%= submit_tag "Log in" %></div>
<% end %>
```

Note that the form uses `form_tag` rather than `form_for` as we're not editing a resource here. The form is submitted to a `sessions_path` and so we'll need a new `SessionsController`.

```
$ rails g controller sessions
```

This controller will have new, `create` and `destroy` actions so that users can log in and out.

/app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  def new
  end

  def create
    if # authenticated?
      session[:user_id] = user.id
      redirect_to root_url, :notice => "Logged in!"
    else
      flash.now.alert = "Invalid email or password"
      render "new"
    end
  end

  def destroy
    session[:user_id] = nil
    redirect_to root_url, :notice => "Logged out!"
  end
end
```

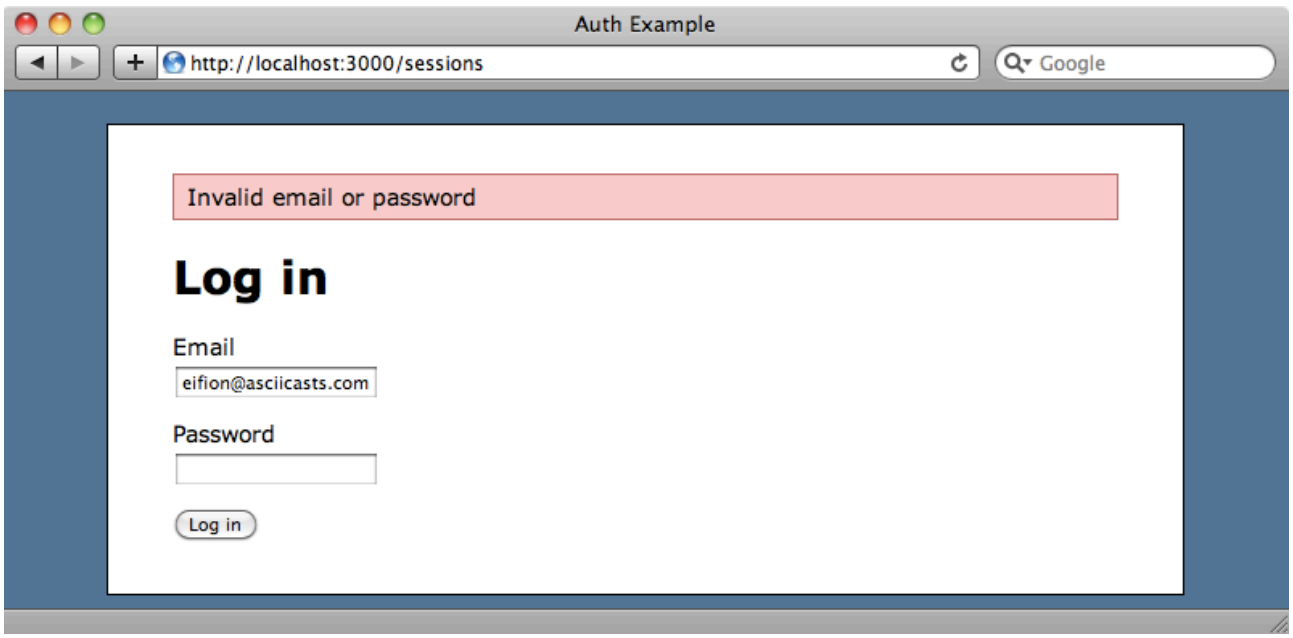
The `create` action isn't quite completed yet as we're yet to write the code that determines whether the user who's logging in is authenticated or not. This is where we can take advantage of what Rails 3.1 gives us with `secure_password`.

The first thing we'll need to do is to fetch the user whose email address matches the value entered in the form. We can then call `authenticate` on that user, which is a method that `has_secure_password` gives us, passing in the password that was entered in the form. This method will check the password against the password digest in the database. If no matching user is found then `find_by_email` will return `nil` so we check that the user exists before authenticating them.

/app/controllers/sessions_controller.rb

```
def create
  user = User.find_by_email(params[:email])
  if user && user.authenticate(params[:password])
    session[:user_id] = user.id
    redirect_to root_url, :notice => "Logged in!"
  else
    flash.now.alert = "Invalid email or password"
    render "new"
  end
end
```

This is all we need to do to authenticate through `secure_password`. We can test this out by trying to log in. If we enter an incorrect password we won't be authenticated.



If we log in with the correct credentials we'll be authenticated and redirected to the home page.

Elsewhere in our application we need to get the currently logged-in user so we'll add a `current_user` method in the `ApplicationController` and make it a helper method so that we can access it in the views. This method will fetch the current user from the session.

/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  private

  def current_user
    @current_user ||= User.find(session[:user_id]) if session
[:user_id]
  end

  helper_method :current_user
end
```

What's really nice about this method of authentication is how simple the `User` model is, just two lines of code.

/app/models/user.rb

```
class User < ActiveRecord::Base
  has_secure_password
  validates_presence_of :password, :on => :create
end
```

By comparison the solution we created in episode 250 had a much more complex `User` model[3]. That said it would be wise to add an `attr_accessible` line to the model so that only the `email`, `password` and `password_confirmation` fields can be set through the user registration form.

/app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create
end
```

[3] https://github.com/ryanb/railscasts-episodes/blob/master/episode-250/auth/app/models/user.rb

Adding HTTPS

If we're handling authentication in our applications we don't really want to send credentials over the wire in plain text so it's a good idea to use SSL and switch over to HTTPS. Prior to Rails 3.1 this had to be done either manually or through a plugin but now there's a much simpler way to implement it.

To restrict a controller so that it has to be accessed with HTTPS we simply need to add a call to the `force_ssl` class method. If want to restrict this only certain actions within a controller we can use the `:only` or `:except` options like we would when using a `before_filter`.

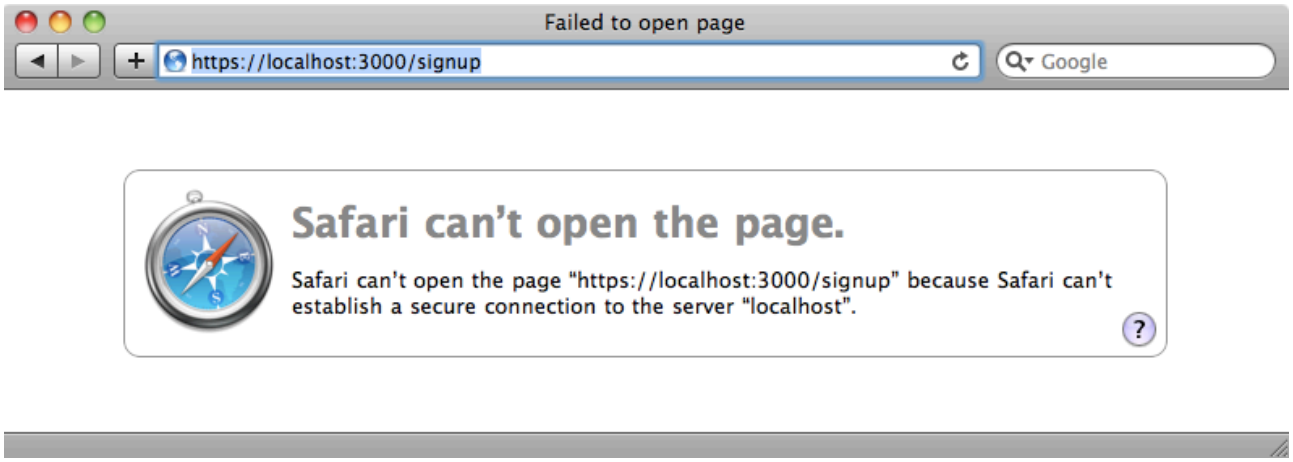
/app/controllers/application_controller.rb

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

  def current_user
    @current_user ||= User.find(session[:user_id]) ↵
      if session[:user_id]
  end

  helper_method :current_user
end
```

The `force_ssl` method only forces HTTPS in test or production modes. If we restart our application's server in production mode and reload the signup page it will try to redirect to the SSL version.



This fails as our server doesn't support HTTPS but if it did we'd see the secure version of this page.

That's it for this episode on authentication in Rails 3.1. All of the new authentication additions it provides make it much easier to add authentication to our Rails applications.