ECE 495/595 – Web Architectures/Cloud Computing

Module 5, Lecture 3: Web Application Security – Access Control

Professor G.L. Heileman

University of New Mexico

Authentication Frameworks

- In the previous lecture we built our own user authentication from scratch.
- In this lecture we'll consider how to build use Rails authentication frameworks, provided as gems.
- The most popular Rails authentication frameworks include: Authlogic, Restful Authentication, Clearance, Devise and Sorcery.
- Next we'll discuss some of the off-the-shelf authentication packages that can be incorporated into a Rails application.
- Finally, we look at how to restrict access to certain parts of a web application depending upon the whether or not a user is logged in, or by the role that the user has.

Devise – Setup

- In this lecture we'll consider the popular Devise authentication framework. Devise is:
  - Rack based;
  - A complete MVC solution based on Rails engines;
  - Allows you to have multiple roles (or models/scopes) signed in at the same time;
  - Based on a modularity concept: use just what you really need.
- In order to use Devise, add the following gem to your Gemfile:

  ```
  gem 'devise'
  ```

  and then run:

  ```
  $ bundle install
  ```

  followed by:

  ```
  $ rails generate devise:install
  ```

  The last command will install devise, and will print some additional setup instructions that you should read. (Note: make sure to remove the HTTP basic authentication from your comments controller.)

Devise – Setup

- The devise install created two files:
  - `config/initializers/devise.rb` – Contains all sorts of configuration options for Devise
  - `config/locales/devise.en.yml` – Contains the messages that Devise uses. You can create messages for additional languages (internationalization) if you'd like.
- Next, since we want to authentication users, we generate a User model using Devise:

  ```
  $ rails generate devise User
  ```

  This generates a model file, a migration, along with a `devise_for` route.
- Take a look at `app/models/user.rb`:

```
class User < ActiveRecord::Base
  # Include default devise modules.  Others available are:
  # :token_authenticatable, :encryptable, :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :email, :password, :password_confirmation, :remember_me
end
```

Devise

Devise is composed of 12 modules (six are used by default):

1. Database Authenticatable – encrypts and stores a password in the database to validate user authenticity during sign in.
2. Token Authenticatable – user sign with authentication token.
3. Omniauthable – adds Omniauth support.
4. Confirmable – sends emails confirmation for sign up.
5. Recoverable – resets the user password and sends reset instructions.
6. Registerable – signing up users through a registration process.
7. Rememberable – remember a user from a saved cookie.
8. Trackable – tracks sign in count, timestamps and IP address.
9. Timeoutable – expires sessions when there is no activity.
10. Validatable – provides validations of email and password.
11. Lockable – locks an account after a specified number of failed sign-in attempts.
12. Encryptable – adds support of other authentication mechanisms besides the default Bcrypt.

Devise – Authentication

- If you want to change the modules that Devise uses by default, you must:
  - Comment/uncomment the appropriate module in `app/models/user.rb`
  - Edit the migration file the Devise created in `db/migrate`.

  We'll stick with the defaults
- Run

  ```
  $ rake db:migrate
  ```

  Now, it you run

  ```
  $ rake routes
  ```

  you'll see that you have a bunch of routes related to user authentication.
- E.g., if you browse to:

  ```
  http://localhost:3000/users/sign_in
  ```

  You'll see that the sign in process is wired up and ready to go. Indeed, we now have a fully functioning authentication system in place. Your previously developed Rspec and Cucumber tests should still pass.

Devise – Authentication

To add navigation links for signing up, signing out, etc. on each page, add this code to `/app/views/layouts/application.html.erb` just before the code that generate displays flash messages:

```
<div id="user_nav">
  <% if user_signed_in?  %>
    Signed in as <%= current_user.email %>.  Not you?
    <%= link_to "Sign out", destroy_user_session_path,
                                             :method => :delete %>
  <% else %>
    <%= link_to "Sign up", new_user_registration_path %> or
    <%= link_to "Sign in", new_user_session_path %>
  <% end %>
</div>
```

Devise – Views

- If you look in your project files, you'll see that the views that Devise is using are note there. They're in the Devise engine.
- If you'd like to pull them into your application, so that you can customize the views, simply run:

  ```
  $ rails generate devise:views
  ```

  This will create view files for you, and copy them into you application directory under `app/views/devise`.
- In addition, we can change the default routes that Devise provides, by editing the `config/routes.rb` file, and using the `:path_names` parameter:

  ```
  devise_for :users, :path_names => { :sign_up => "register",
                                      :sign_in => "login",
                                      :sign_out => "logout" }
  ```

Devise – Restricting Access

- In order to restrict certain actions to users that are logged in, you use a before_filter that Devise provides. E.g.,

```
# app/controllers/posts_controller.rb
before_filter :authenticate_user!, :except => [:index, :show]
```

- Now, if the user is not logged in, and they try to access a restricted page, they will be redirected to the login page.
- The helper functions: user_signed_in?, current_user and user_session are also available for use inside controllers and views.
- Note that some of your tests will now fail unless a default user is created and logs in before each test runs.
- Devise provides test helpers to make it simple to create and log in a default user. Create a file spec/support/devise.rb:

```
RSpec.configure do |config|
  config.include Devise::TestHelpers, :type => :controller
end
```

This will allow you to write controller specs that set up a signed-in user before tests are run.

Devise – Restricting Access

- To make our existing RSpec tests pass, we need to change the `before` method in `specs/controllers/posts_controller_spec.rb`:

```
before(:each) do
  @request.env["devise.mapping"] = Devise.mappings[:user]
  @user = Factory.create(:user)
  sign_in @user
end
```

- In addition, we need the User factory to create valid users. Edit `spec/factories/users.rb` so that it looks like:

```
FactoryGirl.define do
  factory :user do |u|
    u.sequence(:email) { |n| "user#{n}@example.com" }
    u.password "123456"
    u.password_confirmation { |p| p.password }
  end
end
```

- With these changes all of the specs should pass using the Devise authentication.

THE UNIVERSITY of NEW MEXICO

Devise – Restricting Access

- We also need to fix our Cucumber test. Let's change the one scenario that requires authentication to look like:

```
Scenario:  Create Valid Posts
  Given I have no posts
  When I go to the list of posts
  And I sign in
  And I follow New Post
  And I fill in "Title" with "Amazing Post"
  And I fill in "Body" with "omg lol!"
  And I press "Create Post"
  Then I should see "Post was successfully created."
  And I should see "Amazing Post"
  And I should see "omg lol!"
  And I should have 1 post
```

- Now, with the following step definition, you should be able to get the features to pass again:

```
When /^I sign in$/ do
  visit new_user_session_path
  @user = Factory.create(:user)
  fill_in 'user_email', :with => @user.email
  fill_in 'user_password', :with => @user.password
  click_button('Sign in')
  page.should have_content('Signed in successfully.')
end
```

Devise – Role-based Access

- This `before_filter` approach works well if your authorization needs are simple. However, if you'd like to restrict users to their own posts, or have different types of user roles (e.g., administrator, moderator, user), and restrict access according to a user's role, you need to include an additional authorization solution.
- One way to handle this is to create multiple user models in Devise. I.e., you can create on model called `User` and another called `Admin`, and you can configure them differently (e.g., no password recovery for `Admins`).
- Another solution is to use frameworks that are explicitly designed for role-based access control. Declarative Authorization is one solution, and CanCan another very simple framework. Both of these can be built on top of Devise.