User Authentication and Session Control

CITS3403 Web & Internet Technologies

Reference: Ruby et al, Chapters 14 & 20

Slides © 2009 Cara MacNish, and ©2011, 2012 Wei Liu, University of Western Australia.
Step 1: Scaffolding a User Application

depot> rails generate scaffold User name:string email:string password:digest

depot> rake db:migrate

The generated app/models/user.rb

class User < ActiveRecord::Base
    has_secure_password
end

The devise5 and authlogic6 plugins implement user authentication and session management.
Step 3: modify Gemfile

- Then uncomment out the `bcrypt-ruby` gem in the Gemfile:

```ruby
# To use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

- Then

  `depot> bundle install`

- Restart the server if you are using the built-in Webrick server

- Then

  `depot> rake db:migrate`
has_secure_password

• It does two things:
  – it creates two fields: password and password_confirmation and check to see if they match
  – if so, the password is encrypted into a database field: password_digest
Step 2: Modify the User model

class User < ActiveRecord::Base
  attr_accessible :name, :email, :password
  validates :name, :presence => true, :uniqueness => true,
  :length => { :maximum => 50 } 
  validates :name, :presence => true, :uniqueness => true,
  :length => { :maximum => 50 } 
  validates :email, :presence => true, 
  :format => { :with => /\w+@\w+\./w+/, 
  :message => "abc@examples.com" } 
  validates :password, :length => { :within => 6..40 }
  has_secure_password
end
Variables in memory but not in database

- **attr_accessor** is a ruby syntax that creates an instance variable in the memory only but not persistently stored in the database.

- **attr_accessible** is a rails syntax, which isn’t strictly necessary, but it is a really good idea.
  - It tells Rails which attributes of the model can be modified by outside users.
  - It is import to prevent a mass assignment vulnerability (a serious security hole in Rails).
Step 4: the form in the view

- `_form.html.erb`

```html
<fieldset>
  <legend>Register a new user</legend>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </div>
  <div class="field">
    <%= f.label :password %><br /><%= f.password_field :password %>
  </div>
  <div class="field">
    <%= f.label :confirm %><br /><%= f.password_field :password_confirmation %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
</fieldset>
```

![Form Image](image-url)
Session and Page Access Control

• We need to provide a form that allows them to enter their username and password.

• Once they are logged in, we need to record that fact somehow for the rest of their session (or until they log out).

• We need to restrict access to the user specific parts of the application, allowing only people who are logged in to access their own designated areas.
Step 5: Add sessions for each user

depot> rails generate controller Sessions new create destroy

• In sessions_controller.rb

    def new
    end
    def create
      user = User.find_by_name(params[:name])
      if user and user.authenticate(params[:password])
        session[:user_id] = user.id
        redirect_to products_url
      else
        redirect_to login_url,
        :alert => 'Invalid user/password combination'.
      end
    end
    def destroy
      session[:user_id] = nil
      redirect_to login_url, :notice => 'logged out'
    end
Step 6: Edit the views: app/views/sessions/new.html.erb

```erb
<% if flash[:alert] %>
  <p id="notice"><%= flash[:alert] %></p>
<% end %>

<%= form_tag do %>
  <fieldset>
    <legend>Please Log In</legend>
    <div>
      <label for="name">Name:</label>
      <%= text_field_tag :name, params[:name] %>
    </div>
    <div>
      <label for="password">Password:</label>
      <%= password_field_tag :password, params[:password] %>
    </div>
    <div>
      <%= submit_tag "Login" %>
    </div>
  </fieldset>
<% end %>
```
Form Data to controller
Step 7: Restrict Assess to logged-in user only

class ApplicationController < ActionController::Base

  before_action :authorize

  protected

  def authorize

    unless User.find_by_id(session[:user_id])
      redirect_to login_url, notice: "Please log in"
    end

  end

  protect_from_forgery

end

• Note: use skip_before_action :authorize in the controllers for public page that do not need authentication.
Step 8: Add a Log Out Button

- In app/views/products/index.html.erb

```ruby
<%= button_to 'Log out', logout_path, 
  :method => :delete, 
  :confirm => 'Are you sure?', 
  :controller => "sessions" %>
```
Step 9: Manage the Route

- In config/routes.rb, replace the
  get 'sessions/new'
  get 'sessions/create'
  get 'sessions/destroy'

With the following

```ruby
Depo::Application.routes.draw do
  controller :sessions do
    get 'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end
  # Skipped other routes definitions
end
```

Now you can access /depot/sessions/new as /depot/login
Session Management

• Recall that HTTP is stateless
  – each request is independent
  – shopping cart concept requires history

• Solutions (“fake it!”)
  – encode session information in form data on each page and pass to next
  – add session identifier or data to URL (URL rewriting)
    – cookies
      • chunk of data passed to browser
      • when the browser sends a request to the same site it includes the cookie

• Rails’ two-step solution
  – A 32-hex session id on the client ($16^{32}$)
  – A persistent store of session data indexed by the session id.
Session Management

• Rails provides an abstraction from cookie detail
  ‣ hash-like collection called session
  ‣ any key/value pairs you store in the hash during processing of a request will be available during subsequent requests from same browser

• Storing session information
  1. Rails default is to store all information in cookies. Issues...
     • robustness - eg system problem at browser end, access from different machine (or phone, etc...)
     • security - private information stored at browser end?
  2. File on server. Issues...
     • load sharing - popular website requests may be distributed among multiple servers
  3. Database
     • secure, persistent, available to all servers,...
Putting Sessions in the Database

• First, create a migration containing session table definition. Shortcut provided...

depot> rake db:sessions:create
Or in Rails 3.2
depot> rails generate session_migration

    active_record
    create   db/migrate/20121015135117_add_sessions_table.rb

In Rails 4, the ActiveRecord session store has been extracted out of Rails into it's own gem.
Modify your Gemfile with:
gem 'activerecord-session_store'
depot> bundle install
depot> rails generate active_record:session_migration

• Then, apply migration to create “sessions” table...

depot> rake db:migrate
Putting Sessions in the Database

• Next, tell Rails to use database storage for sessions (default is to store everything in cookies)
  
  – configuration option
  – uncomment line in
    
    `depot/initializer/session_store.rb`

```
# Use the database for sessions instead of the cookie-based default,  
# which shouldn't be used to store highly confidential information  
# (create the session table with "rails generate session_migration")
Depot::Application.config.session_store  :active_record_store
```
The following slides are not assessed

- They are steps used for user authentication in earlier versions of RoR

Authentication in Rails 3.2
Step 1: Scaffolding a User Application

depot> rails generate scaffold User name:string email:string password_digest:string

depot> rake db:migrate

The generated app/models/user.rb

class User < ActiveRecord::Base
  attr_accessible :name, :password_digest
end
Step 2: Modify the User model

class User < ActiveRecord::Base
  attr_accessible :name,:password,:email,:password_confirmation

  validates :name, presence => true, uniqueness => true
  length => { :maximum => 50 }
  validates :email, presence => true,
  format => { :with => /\w+@\w+\.\w+/,
              :message =>"abc@examples.com" }

  # Automatically check for password presence and
  # match the attribute password_confirmation with password.

  validates :password, length => { :within => 6..40 }
  validates :password_confirmation, presence => true
  has_secure_password

end
Variables in memory but not in database

- `attr_accessor` creates an instance variable in the memory only but not persistently stored in the database

- `attr_accessible` isn’t strictly necessary, but it is a really good idea.
  - It tells Rails which attributes of the model can be modified by outside users.
  - It is import to prevent a mass assignment vulnerability (a serious security hole in Rails)
Overriding validation error message

• By default, rails will complain that “Password digest can't be blank” if password field is left blank

• To override this, add the following into config/locales/en.yml file, add the following:

```yaml
en:
  activerecord:
    attributes:
      user:
        password_digest: "Password"
```
The following slides are not assessed

- They are steps used for user authentication in earlier versions of RoR

Authentication in Rails 2.x
require 'digest/sha2' # Loading the encryption algorithm

class User < ActiveRecord::Base
  attr_accessor :password
  attr_accessible :name, :email, :password,
                 :password_confirmation

  validates :name, :presence => true,
            :length => { :maximum => 50 }
  validates :email, :presence => true,
            :format => { :with => /\w+@\w+\.\w+/,
                         :message => "abc@examples.com" }

  # Automatically create the virtual attribute
  # 'password_confirmation'.
  validates :password, :presence => true,
             :confirmation => true,
             :length => { :within => 6..40 }

  # Database callback
  before_save :encrypt_password

  # Add more on encryption later ...
end
private
  def encrypt_password
    self.salt = make_salt if new_record?
    self.encrypted_password = encrypt(password)
  end

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

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

  def secure_hash(string)
    Digest::SHA2.hexdigest(string)
  end
Step 2-3: Define authenticate method in app/models/user.rb

```ruby
def authenticate?(submitted_password)
    encrypted_password == encrypt(submitted_password)
end
```
Step 2-4: Store salt and encrypted password

```
>> rails generate migration add_password_to_users
encrypted_password:string

>> rails generate migration add_salt_to_users salt:string

>> rake db:migrate
```