User Authentication

In this handout we will discuss about the user authentication in Ruby on Rails (RoR), with enabled Bootstrap from the previous handout. Go to the railsapp folder and create a new rails project

```
$ cd  <your rails app folder>
$ rails new userauth -d postgresql
$ cd userauth
```

Lets generate user controller

```
$ rails generate controller users new
```

Create user model with attribute **name** and **email**.

```
$ rails generate model User name:string email:string
$ rake db:create
$ rake db:migrate
```

Working with rails console

Rails console is the tool of choice for exploring data models in console. If we do not want to make changes to our database, we will start the console in a **sandbox**, with this flag the changes you make to the database will not be permanent, it will rollback on exiting the console.

```
$ rails console --sandbox
```

Let's add a user to the database using console

```
>> user = User.new
>> user = User.new (name: "Hamid Khan", email: "hamid@gamil.com")
```

To check the entry we made is valid for the user

```
>> user.valid?
>> user.save
```

To check the attributes of the users

```
>> user.valid?
>> user.save
```

It is often convenient to make and save a model in two steps as we have here, but Active Record also lets you combine them into one step with **User.create**:

```
>> User.create(name: "Waleed Khan",email: "waleed@example.org")
```

Adding validations to the User model

Add the following validation in user model

app/models/user.rb

```
class User <ActiveRecord::Base
  validates :name, presence: true
end
```

Now check the validation of user data in rails console

```
    $ rails console --sandbox
    >> user = User.new(name: "", email: "hamid@example.com")
    >>user.valid?
    =>false
```

Check the error messages

```
  >>user.errors.full_messages
  =>["Name can't be blank"]
  >> user.save
  =>  false
```

Adding validation for email, first check that email field is not empty

```
class User <ActiveRecord::Base
  validates :name, presence: true
  validates :email,  presence: true
end
```

Length validation

We have constrained our User model to require a name for each user, but we should go further and constraint the length of the user name field and email address.

```
class User <ActiveRecord::Base
  validates :name,presence: true,length: { maximum: 50 }
  validates :email,presence: true,length: { maximum: 255 }
end
```

Format validation

Email address needs some more constraints such as format validation. For the email format validation we can write a command

```
  validates :email,format: { with: /<regular expression>/ }
```

and we write a constant

```
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
```

Now how the **user.rb** file will look like.

Expression	Meaning
/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i	full regex
/	start of regex
\A	match start of a string
[\w+\-.]+	at least one word character, plus, hyphen, or dot
@	literal "at sign"
[a-z\d\-.]+	at least one letter, digit, hyphen, or dot
\.	literal dot
[a-z]+	at least one letter
\z	match end of a string
/	end of regex
i	case-insensitive

Figure 1: Breaking down the valid email regex.

```
class User <ActiveRecord::Base
  validates :name,presence: true,length:  maximum: 50
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email,presence: true,length: { maximum: 255 },
      format: { with: VALID_EMAIL_REGEX }
end
```

Uniqueness validation

To enforce uniqueness of email addresses (so that we can use them as usernames), we'll be using the :unique option to the validates method. Update the email validation in user model.

app/model/user.rb

```
validates :email,presence: true,length: { maximum: 255 },
      format: { with: VALID_EMAIL_REGEX },
      uniqueness: true
```

Email addresses are typically processed as if they were case- insensitive—that is, foo@bar.com is treated the same as FOO@BAR.COM or FoO@BAr.coM—so our validation should incorporate this as well

app/model/user.rb

```
validates :email,presence: true,length: { maximum: 255 },
      format: { with: VALID_EMAIL_REGEX },
      uniqueness: { case_sensitive: false }
```

There's just one small problem: The the Active Record uniqueness validation does not guarantee uniqueness at the database level. Here's a scenario that explains why:

1. Alice signs up for the sample app, with address alice@wonderland.com.
2. Alice accidentally clicks on "Submit" twice, sending two requests in quick succession.
3. The following sequence occurs: Request 1 creates a user in memory that passes validation, request 2 does the same, request 1's user gets saved, request 2's user gets saved.
4. Result: Two user records with the exact same email address exist, despite the uniqueness validation.

We just need to enforce uniqueness at the database level as well as at the model level. Our method is to create a database index on the email column

```
$ rails generate migration add_index_to_users_email
```

Go and check the newly created migration file and the following line.

```
add_index :users, :email
```

The User model will look like now

app/model/user.rb

```
class User <ActiveRecord::Base
  before_save {self.email = email.downcase }
  validates :name,presence: true,length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email,presence: true,length: { maximum: 255 },
    format: { with: VALID_EMAIL_REGEX },
    uniqueness: { case_sensitive: false }
end
```

Adding a secure password

Now that we've defined validations for the name and email fields, we're ready to add the last of the basic User attributes: a secure password. The method is to require each user to have a password (with a password confirmation), and then store a hashed version of the password in the database.

A Hashed password

When you include **has_secure_password** in the user model, we receive the following functionality.

- The ability to save a securely hashed password_digest attribute to the database
- A pair of virtual attributes (password and password_confirmation), including presence validations upon object creation and a validation requiring that they match
- An authenticate method that returns the user when the password is correct (and false otherwise)

Lets add an attribute password_digest in the user table

```
$ rails generate migration add_password_digest_to_users password_digest:string
```

Check the migrate file for your understanding and migrate the database.

```
$ rake db:migrate
```

To make the password digest, has_secure_password uses a state-of-the-art hash function called bcrypt. Add a **bcrypt** gem into your **Gemfile**.

Gemfile

```
gem 'bcrypt','3.1.7'
```

and then to install gem type

Gemfile

```
bundle install
```

Minimum password length and has_secure_password

Add the minimum password length and add a method has_secure_password in the user model.

app/model/user.rb

```
class User <ActiveRecord::Base
  before_save {self.email = email.downcase }

  validates :name,presence: true,length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email,presence: true,length: { maximum: 255 },
      format: { with: VALID_EMAIL_REGEX },
      uniqueness: { case_sensitive: false }

  has_secure_password
  validates :password,length: { minimum: 6 }
end
```

Creating and authenticating a User

Now that the basic User model is complete, we'll create a user in the database as preparation for making a page to show the user's information

```
$ rails console
>>User.create(name: "Waleed",email: "waleed@example.com",
?>   password: "foobar",password_confirmation: "foobar")
```

To check the user and its password in rails console

```
>>user = User.find_by(email: "waleed@example.com")
>>user.password_digest
=>"$2a$10$YmQTuuDNOszvu5yi7auOC.F4G//FGhyQSWCpghqRWQWITUYlG3XVy"
```

```
>>user.authenticate("my_password")
false
>>user.authenticate("foobaz")
false
```

Here user.authenticate returns false, indicating an invalid password. If we instead authenticate with the correct password, **authenticate** returns the corresponding user itself:

```
>>user.authenticate("foobar")
```

!! converts an object to its corresponding boolean value.

```
>>!!user.authenticate("foobar")
```

User sign up

In this section, we will discuss about the user sign up to the application. Lets add first the sign-up and the user resource to the routes.rb file

```
get 'signup' =>'users#new'
resources :users
```

Sign-up Form

In this section we will create a sign up form for the user to register.

User show

First create an user in the rails console and then create a show action and show view page as follows

app/conrollers/users_controller.rb

```
    def show
      @user = User.find(params[:id])
    end
```

and the view page (*views/users/show.html.erb*)

show.html.erb

```
  <%= @user.name %> <br>
  <%= @user.email %>
```

and add a new page looks like now (*views/users/new.html.erb*)

```
    def new
      @user = User.new
    end
```

User new

Let's create the user new action and view page for the signup form.

app/views/users/new.html.erb

```
<h1> Sign up </h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control'%>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation,"Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%=  f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Let's add some SCSS code for the signup form.

app/assets/stylesheets/custom.css.scss

```
.
.
.
/* forms */
input,textarea,select,.uneditable-input
    border: 1px solid #bbb;
    width: 100%;
    margin-bottom: 15px;
    @include box_sizing;

input
    height: auto !important;
```

Lets include a Sass mixin.

app/assets/stylesheets/custom.css.scss

```
/* mixins,variables,etc. */
$gray-medium-light: #eaeaea;

@mixin box_sizing {
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
   box-sizing: border-box;
}
```

Now we will add a create action to save the new user form data.

```
def create
  @user = User.new(user_params)
  if @user.save
    redirect_to @user
  else
    render 'new'
  end
end

private
  def user_params
    params.require(:user).permit(:name,:email,:password,
        :password_confirmation)
    end
```

A flash message

We need a message for short time that user is successfully saved.

```
def create
  @user = User.new(user_params)
    if @user.save
      flash[:success] = "User successfully created!"
      redirect_to @user
    else
      render 'new'
    end
  end
```

Now add the message in application.html.erb

```
<% flash.each do |message_type,message| %>
    <div class="alert alert-<%= message_type %>">
      <%= message %>
    </div>
<% end %>
```

Sign up error messages

Lets create a folder *shared* under views and keep *error_messages* partials for displaying error massages on sign up

app/views/shared/_error_massages.html.erb

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <div class="alert alert-danger">
      The form contains
      <%= pluralize(@user.errors.count,"error") %>
    </div>
    <ul>
      <% @user.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

and call the this _error_messages partials in the new file.

app/views/users/new.html.erb

```
<h1> Sign up </h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control'%>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation,"Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%=  f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

There is an CSS id **error_explanation** for use in styling the error messages. In addition, after an invalid submission Rails automatically wraps the fields with errors in divs with the CSS class **field_with_errors**.

app/assets/stylesheets/custom.css.scss

```
.
.
.
/* forms */
.
.
.
#error_explanation {
  color:  red;
    ul  {
      color: red;
      margin: 0 0 30px 0;
    }
}
.field_with_errors {
  @extend .has-error;
    .form-control {
      color: $state-danger-text;
    }
}
```

Log In, Log out

Sessions controller

The elements of logging in and out correspond to particular *REST* actions of the Sessions controller. The login form is handled by the *new* action, actually logging in is handled by sending a *POST* request, and logging out is handled by sending a *DELETE* request to the destroy action.

We'll generate a sessions controller with a new action:

```
$ rails generate controller Sessions new
```

Adding a resource to get the standard *RESTful* actions for sessions. *config/routes.rb*

```
get 'login' => 'sessions#new'
post 'login' =>'sessions#create'
delete 'logout' =>'sessions#destroy'
```

Now check the routes by typing *rake routes*

Login form

The main difference between the session form and the sign-up form is that we do not need to have no Session model, Code for the login form.

app/views/session/new.html.erb

```
<h1>Log in</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session,url: login_path) do |f| %>
      <%= f.label :email %>
      <%= f.email_field :email,class: 'form-control' %>
      <%= f.label :password %>
      <%= f.password_field :password,class: 'form-control' %>
      <%= f.submit "Log in",class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
<p> New user? <%= link_to "Sign up now!",signup_path %></p>
```

Finding and Authenticating a User

The first step in creating sessions (logging in) is to handle invalid input. We will start by reviewing what happens when a form gets submitted, and then arrange for helpful error messages to appear in the case of login failure

app/controllers/sessions_controller.rb

```
def new
end

def create
  user = User.find_by(email: params[:session][:email].downcase)

  if user && user.authenticate(params[:session][:password])
    # Log the user in and redirect to the user's show page.
  else
    # Create an error message.
    render 'new'
  end
end

def destroy
end
```

Rendering with a Flash Message

app/controllers/sessions_controller.rb

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    # Log the user in and redirect to the user's show page.
  else
    flash[:danger] = 'Invalid email/password combination' # Not quite right!
    render 'new'
  end
end
```

Logging In

Now that our login form can handle invalid submissions, the next step is to handle valid submissions correctly by actually logging a user in. In this section, we will log the user in with a temporary session cookie that expires automatically upon browser close. Later we will add sessions that persist even after closing the browser.

A sessions helper module is generated automatically when generating the sessions controller. Moreover, such helpers are automatically included in Rails views; by including the module into the base class of all controllers (the Application controller), we arrange to make them available in our controllers as well.

app/controllers/application_controller.rb

```
class ApplicationController <ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper
end
```

The log_in form
Let's consider the following command provided by rails

session[:user_id] = user.id

This places a temporary cookie on the user's browser containing an encrypted version of the user's ID, which allows us to retrieve the ID on subsequent pages using session[:user_id]. In contrast to the persistent cookie created by the cookies method, the temporary cookie created by the session method expires immediately when the browser is closed.

Because we want to use the same login technique in several different places, we'll define a method called *log_in* in the Sessions helper.

app/helpers/session_helper.rb

```
module SessionsHelper
  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
end
```

Now we can update our create action in session controller to call the *log_in* method on the successful login.

app/controllers/session_controller.rb

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end
```

Current User

We add the *current_user* method into the session helper to retrieve the current user on every page.

app/helpers/session_helper.rb

```
.
.
# Returns the current logged-in user (if any).
def current_user
  @current_user ||= User.find_by(id: session[:user_id])
end
.
.
```

Changing the layout links

The first practical application of logging in involves changing the layout links based on login status. Let's add a method to test whether the user is logged in or not.

app/helpers/session_helper.rb

```
.
.
# Returns true if the user is logged in,false otherwise.
def logged_in?
  !current_user.nil?
end
.
.
```

Lets update the header partials to include

app/views/layouts/application.html.erb

```
.
.
<% if logged_in? %>
  <%= link_to "Log out",logout_path,method: "delete" %>
<% else %>
  <%= link_to "Log in",login_path %>
<% end %>
.
.
```

Login upon sign-up

The newly registered users are no automatically logged in by the system. We integrate now that it automatically logs in the user who registers.

app/controller/users_controller.rb

```
.
.
def create
  @user = User.new(user_params)

  if @user.save
    log_in @user
    flash[:success] = "User successfully logged in!"
    redirect_to @user
  else
    render 'new'
  end
end
.
.
```

Logging Out

Logging out involves undoing the effects of the log_in method, which involves deleting the user ID from the session. we have a command
session.delete(:user_id)
Lets add a method to the sessions_helper.rb

app/helpers/sessions_helper.rb

```
.
.
# Logs out the current user.
def log_out
  session.delete(:user_id)
  @current_user = nil
end
.
.
```

Now we also add a destroy method in session controller

app/controllers/sessions_controller.rb

```
.
.
def destroy
  log_out
  redirect_to root_url
end
.
.
```

Updating, Showing, and Deleting Users

Updating Users

Lets first add an edit and update action to the user controller

app/controllers/users_controllers.rb

```
def edit
  @user = User.find(params[:id])
end
```

The edit form

```
<h1>Update your profile</h1>
<%= form_for(@user) do |f| %>
    <%= render 'shared/error_messages' %>
    <%= f.label :name %>
    <%= f.text_field :name,class: 'form-control' %>

    <%= f.label :email %>
    <%= f.email_field :email, class: 'form-control' %>

    <%= f.label :password %>
    <%= f.password_field :password, class: 'form-control' %>

    <%= f.label :password_confirmation,"Confirmation" %>
    <%= f.password_field :password_confirmation, class: 'form-control' %>

    <%= f.submit "Save changes",class: "btn btn-primary" %>
<% end %>
```

and the update action

```
def update
  @user = User.find(params[:id])

  if @user.update_attributes(user_params)
    flash[:success] = "Profile updated"
    redirect_to @user
  else
    render 'edit'
  end
end
```

Requiring Logged-in Users

Before we edit the user we need to enforce the user to log in. Open the User controller and add.

```
class UsersController <ApplicationController
  before_action :logged_in_user,only: [:edit,:update]
.
.
.
  private

    # Confirms a logged-in user.
      def logged_in_user
        unless logged_in?
          flash[:danger] = "Please log in."
          redirect_to login_url
        end
      end
end
```

Requiring the right user

Of course, requiring users to log in is not quite enough; users should be allowed to edit only their own information.
To redirect users trying to edit another user's profile, we'll add a second method called **correct_user**, together with a before filter to call it.

```
class UsersController <ApplicationController
  before_action :logged_in_user,only: [:edit,:update]
  before_action :correct_user,only: [:edit,:update]


private

  def correct_user
    @user = User.find(params[:id])
    # redirect_to(root_url) unless current_user?(@user)
    redirect_to(root_url) unless @user == current_user
  end
end
```

Friendly Forwarding

Our site authorization is complete as written, but there is one minor blemish: When users try to access a protected page, they are currently redirected to their profile pages regardless of where they were trying to go. In other words, if a non-logged-in user tries to visit the edit page, after logging in the user will be redirected to /users/1 instead of /users/1/edit. It would be much friendlier to redirect users to their intended destination instead.

```
def redirect_back_or(default)
  redirect_to(session[:forwarding_url] || default)
  session.delete(:forwarding_url)
end

# Stores the URL trying to be accessed.
def store_location
  session[:forwarding_url] = request.url if request.get?
end
```

Open the user controller and edit the logged_in_user method to store the older location you are currently stand

```
def logged_in_user
  unless logged_in?
    store_location
    flash[:danger] = "Please log in."
    redirect_to login_url
  end
end
```

and restore the older location from the create action in user controller

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    log_in user
    redirect_back_or user
  else
    flash.now[:danger] = 'Invalid email/password combination'
    render 'new'
  end
end
```