[Ruby on Rails: Authentication](https://www.codecademy.com/en/courses/rails-auth)

Many web apps let users sign up for a new account and log in and out of their accounts. Together, signing up, logging in and logging out make up an authentication system.

Let's create an authentication system for a photosharing website built with Ruby on Rails.

1. We've provided a Rails app to get you started. Let's take a look - first install the gems in the **Gemfile**.

$ bundle install

1. Next start a Rails development server.

$ rails server

**3.** Then visit [http://localhost:8000](http://localhost:8000/) to see a list of all albums.

**4.** Click on an album to see the photos inside it.

How can we add an authentication system to this app? Click Next to learn more.

Using the [request/response cycle](https://www.codecademy.com/articles/request-response-cycle-forms) as a guide, here's how authentication fits in:

**Turn one:**

1. When a user visits the signup page, the browser makes an HTTP GET request for the URL /signup.
2. The Rails router maps the URL /signup to the Users controller's newaction. The new action handles the request and passes it on to the view.
3. The view displays the signup form.

**Turn two:**

1. When the user fills in and submits the form, the browser sends the data via an HTTP POST request to the app.
2. The router maps the request to the Users controller's create action.
3. The create action saves the data to the database and redirects to the albums page. The action also creates a new *session*.

What is a session? A session is a connection between the user's computer and the server running the Rails app. A session starts when a user logs in, and ends when the user logs out.

**5.** Looking at the request/response cycle, we need five parts to add signup machinery to the app: a model, a controller, routes, views, and logic for sessions. Let's start here by creating a model.

Generate a model named User.

**6.** In **app/models/user.rb**, add a method namedhas\_secure\_password.

class User < ActiveRecord::Base

has\_secure\_password

end

**7.** In the **Gemfile** on line 30, uncomment the bcrypt gem

**8.** Install the gems.

What did we just do?

1. We created a model named User.
2. In the model, we used the method has\_secure\_password. This method adds functionality to save passwords securely.
3. In order to save passwords securely, has\_secure\_password uses an algorithm called bcrypt. To use bcrypt, we added the bcrypt gem to the Gemfile.

Now that the User model is set up, let's continue by adding columns to the migration files.

**9.** Open the migration file in **db/migrate/** for the users table and add:

* a string column called first\_name
* a string column called last\_name
* a string column called email
* a string column called password\_digest

**10.** Run a migration to update the database.

Nice work! You've added columns to the users table and ran a migration to update the database.

What's the password\_digest column for? When a user submits her password, it's not a good idea to store that password as is in the database; if an attacker somehow gets into your database, he would be able to see all your users' passwords.

One way to defend against this is to store passwords as encrypted strings in the database. This is what the has\_secure\_password method helps with - it uses the bcrypt algorithm to securely hash a user's password, which then gets saved in the password\_digest column.

Then when a user logs in again, has\_secure\_password will collect the password that was submitted, hash it with bcrypt, and check if it matches the hash in the database.

**11.** Now that the models are set up, let's move on to the rest of the request/response cycle and create the controllers, routes, and views needed for the signup machinery.

Generate a controller named Users.

**12.** In the routes file, add these routes:

get 'signup' => 'users#new'

resources :users

**13.** Next, in the Users controller add the new action.

def new

@user = User.new

end

**14.** Then in **app/views/users/new.html.erb**, on line 7, use form\_for to create a form with the fields of the @user object.

We've provided CSS in **app/assets/stylesheets/application.css**.

**15.** Start the Rails server.

Then visit <http://localhost:8000/signup> to preview the signup page in the browser.

The form won't work just yet, we'll finish it up next.

Great! When you visit the URL /signup, the browser makes a GET request for the URL. This request hits the Users controller's new action, which returns a view displaying the signup page.

**16.** Next, let's take in data submitted through the signup form and save it to the database.

In the Users controller, add a private method user\_params

Private

def user\_params

params.require(:user).permit(:first\_name, :last\_name, :email, :password)

end

**17.** Between the new action and the private method, add the create action

def create

@user = User.new(user\_params)

if @user.save

session[:user\_id] = @user.id

redirect\_to '/'

else

redirect\_to '/signup'

end

end

**18.** Visit <http://localhost:8000/signup> and sign up as a new user.

Nice job! Now when you fill in the signup form and submit it, the data is sent to the Rails app via a POST request. The request hits the User controller's createaction. The create action saves the data, creates a new session, and redirects to the albums page.

How is a new session created? Sessions are stored as key/value pairs. In thecreate action, the line

session[:user\_id] = @user.id

creates a new session by taking the value @user.id and assigning it to the key:user\_id.

**19.** Let's check whether the data saved to the database using the *Rails console*. The Rails console is a useful tool to interact with Rails apps. We'll use it here to query the database.

Start the Rails console by running

$ rails console

**20.** Retrieve all signed up users in the database by typing

> User.all

Your information should show up in the results.

**21.** Visit <http://localhost:8000/signup> to sign up a few more users through the signup form.

Use the Rails console to retrieved all signed up users again.

**22.** Exit the Rails console by typing exit.

Now that users can sign up for a new account, let's add the ability to log in and log out of the app. Using the request/response cycle as a guide again, here's how logging in and logging out fits in.

**Turn one:**

1. When the user visit the login page, the browser makes a GET request for the URL /login.
2. The Rails router maps the URL /login to the Sessions controller's newaction. The new action handles the request and passes it on to the view.
3. The view displays the login form.

**Turn two:**

1. When the user fills in and submits the form, the browser sends the data via a POST request to the app.
2. The router maps the request to the Sessions controller's create action.
3. The create action verifies that the user exists in the database. If the user exists, the create action logs the user in by creating a new session. Otherwise, it reloads the login page.

**23.** Let's begin by adding a login page. Looking at the request/response cycle, we need five parts to add login machinery to the app: a model, a controller, routes, views, and logic for sessions. As we already created the User model when building the signup page, let's start here by creating a controller.

Generate a controller named Sessions.

**24.** In the routes file, create a route that maps requests for the URL '/login' to the Sessions controller's new action.

**25.** In the Sessions controller, add the new action

def new

end

**26.** Then in **app/views/sessions/new.html.erb**, on line 7, use form\_forto create a login form:

<%= form\_for(:session, url: login\_path) do |f| %>

<%= f.email\_field :email, :placeholder => "Email" %>

<%= f.password\_field :password, :placeholder => "Password" %>

<%= f.submit "Log in", class: "btn-submit" %>

<% end %>

We've also provided CSS in **app/assets/css/application.css**.

**27.** Restart the Rails server and visit <http://localhost:8000/login> in the browser.

The form won't work just yet. We'll finish it up next.

Well done!

1. When you visit the URL /login, the browser makes a GET request for the URL. This request hits the Sessions controller's new action, which returns a view displaying the login page.
2. In the login form, we use form\_for(:session, url: login\_path) do |f|. This refers to the name of the resource and corresponding URL. In the signup form, we used form\_for(@user) do |f| since we had a User model. For the login form, we don't have a Session model, so we refer to the parameters above.

**28.** Next, let's take in data submitted through the form and log the user in by starting a new session.

In the routes file, create a route that maps the URL '/login' to the Sessions controller's create action.

post 'login' => 'sessions#create'

**29.** In the Sessions controller, add the create action

def create

@user = User.find\_by\_email(params[:session][:email])

if @user && @user.authenticate(params[:session][:password])

session[:user\_id] = @user.id

redirect\_to '/'

else

redirect\_to 'login'

end

end

**30.** Visit <http://localhost:8000/login> and log in with your email and password.

Nice job! Now when you fill in the login form and submit it, the data is sent to the Rails app via a POST request. The request hits the Sessions controller's createaction. The create action checks whether your email and password exist in the database, creates a new session, and redirects to the albums page.

**31.** Now that users can log in, let's add the ability to log out.

In the routes file, create a route that maps the URL /logout to the Sessions controller's destroy action:

delete 'logout' => 'sessions#destroy'

**32.** In the Session controller, add the destroy action by setting the session hash to nil and redirecting to the root path

def destroy

session[:user\_id] = nil

redirect\_to '/'

end

Great work so far! We've built an authentication system that lets new users sign up for the site, and lets existing users log in and out.

However, there's one problem - even after you log out, you can still access the albums page. Why does this happen? Let's look at the request/response cycle:

1. Currently when a user visits the URL /albums, the browser first makes a request for that URL.
2. The request hits the Rails router.
3. The router sends the request to the Albums controller's index action*regardless of whether a user is logged in*.

What we want instead is for only users who are logged in to see the albums page; otherwise they should be redirected to the login page. This means that we need to check whether a user is logged in before sending her request on to the Albums controller's index action. Let's see how to do this.

**33.** In **app/controllers/application\_controller.rb**, add a method named current\_user

helper\_method :current\_user

def current\_user

@current\_user ||= User.find(session[:user\_id]) if session[:user\_id]

end

**34.** Below current\_user, add another method named require\_user:

def require\_user

redirect\_to '/login' unless current\_user

end

How do these methods work?

1. The current\_user method determines whether a user is logged in or logged out. It does this by checking whether there's a user in the database with a given session id. If there is, this means the user is logged in and@current\_user will store that user; otherwise the user is logged out and@current\_user will be nil.
2. The line helper\_method :current\_user makes current\_user method available in the views. By default, all methods defined in Application Controller are already available in the controllers.
3. The require\_user method uses the current\_user method to redirect logged out users to the login page.

Here's more on the [||= syntax](http://stackoverflow.com/questions/995593/what-does-or-equals-mean-in-ruby). Here's more on the [unless keyword](https://signalvnoise.com/posts/2699-making-sense-with-rubys-unless).

**35.** Let's use require\_user in the Albums controller in order prevent logged out users from accessing these pages. In the Albums controller, add this as the first line inside the class:

before\_action :require\_user, only: [:index, :show]

The before\_action command calls the require\_user method before running the index or show actions.

**36.** Let's use current\_user in application layout to update the nav items depending on whether a user is logged in or out. In**app/views/layouts/application.html.erb**, within<div class="nav pull-right"> on line 21 add:

<% if current\_user %>

<ul>

<li><%= current\_user.email %></li>

<li><%= link\_to "Log out", logout\_path, method: "delete" %></li>

</ul>

<% else %>

<ul>

<li><%= link\_to "Login", 'login' %></a></li>

<li><%= link\_to "Signup", 'signup' %></a></li>

</ul>

<% end %>

**37.** Log out of the app, and then visit <http://localhost:8000/albums> in the browser. You should be be redirected to the login page.

**38.** Log in to the app with your email and password, and then visit <http://localhost:8000/albums>. You should now be able to access it.

Congratulations! You built an authentication system from scratch. What can we generalize so far?

* An authentication system is made up of sign up, log in, log out functionality.
* The password\_digest column and has\_secure\_password method are provided by bcrypt to store passwords securely.
* A *session* begins when a users logs in, and ends when a user logs out.
* The current\_user method allow us to access the current user;require\_user redirects to the root of the app if there is no such user.
* Before actions act as filters. They call methods before executing controller actions.