(/)

Ruby(https://www.sitepoint.com/ruby/) - June 16, 2016 - By Ilya Bodrov-Krukowski (https://www.sitepoint.com/author/ibodrov/)

Create a Chat App with Rails 5, ActionCable, and Devise

Rails 5 has introduced a bunch of new great features, but one of the most anticipated ones is of course ActionCable (https://github.com/rails/rails/tree/master/actioncable). ActionCable seamlessly integrates WebSockets into your application and offers both client-side JS and server-side Ruby frameworks. This way, you can write real-time features in the same styles as the rest your application, which is really cool.

Learn more on ruby with our tutorial Simulate User Behavior and Test Your Ruby App (https://www.sitepoint.com/premium/screencasts/feature-tests-with-rspec-simulate-user-behavior-and-test-your-ruby-app) on SitePoint.

Some months ago I wrote a series of articles describing how to build mini-chat with Rails using AJAX (https://www.sitepoint.com/mini-chat-rails/), WebSockets powered by Faye (https://www.sitepoint.com/realtime-mini-chat-rails-faye/) and Server-sent events (https://www.sitepoint.com/mini-chat-rails-server-sent-events/). Those articles garnered some attention, so I decided to pen a new part of this series instructing how to use ActionCable to achieve the same goal.

This time, however, we will face a bit more complicated task and discuss the following topics:

- Preparing application and integrating Devise
- Introducing chat rooms
- Setting up ActionCable
- Coding client-side
- Coding server-side with the help of background jobs
- Introducing basic authorization for ActionCable
- Preparing application to be deployed to Heroku

The source code can be found at GitHub (https://github.com/bodrovis/Sitepoint-source/tree/master/Chat_with_ActionCable_and_Devise).

The working demo is available at sitepoint-actioncable.herokuapp.com (https://sitepoint-actioncable.herokuapp.com).

Preparing the Application

Start off by creating a new application. Support for ActionCable was added only in Rails 5, so you will have to use this version (currently 5.0.0.rc1 is the latest one):

```
$ rails new CableChat -T
```

Now add a couple of gems:

Gemfile

```
[...]
gem 'devise'
gem 'bootstrap', '~> 4.0.0.alpha3'
[...]
```

Devise (https://github.com/plataformatec/devise) will be used for authentication and authorization (you may read this article (https://www.sitepoint.com/devise-authentication-in-depth/) to learn more) and Bootstrap 4 (https://github.com/twbs/bootstrap-rubygem) – for styling.

Run

```
$ bundle install
```

Add Bootstrap's styles:

stylesheets/application.scss

```
@import "bootstrap";
```

Run the following commands to install Devise, generate a new `User` model, and copy views for further customization:

```
$ rails generate devise:install
$ rails generate devise User
$ rails generate devise:views
$ rails db:migrate
```

Now restrict access to all pages of the site to authenticated users only:

application_controller.rb

```
[...]
before_action :authenticate_user!
[...]
```

Chat Rooms

The next step is to add support for chat rooms, so generate the following model:

```
$ rails g model ChatRoom title:string user:references
$ rails db:migrate
```

A chat room should have a creator, so make sure you establish a one-to-many relation between `chat_rooms` and `users`:

models/chat_room.rb

```
[...]
belongs_to :user
[...]
```

models/users.rb

```
(/)
[...]
has_many :chat_rooms, dependent: :destroy
[...]
```

Code a controller to list and create chat rooms:

chat_rooms_controller.rb

```
class ChatRoomsController < ApplicationController
  def index
    @chat_rooms = ChatRoom.all
  end

  def new
    @chat_room = ChatRoom.new
  end

  def create
    @chat_room = current_user.chat_rooms.build(chat_room_params)
    if @chat_room.save
      flash[:success] = 'Chat room added!'
      redirect_to chat_rooms_path
    else
      render 'new'
    end
  end

  private

  def chat_room_params
    params.require(:chat_room).permit(:title)
  end
end
```

Now a bunch of really simple views:

views/chat_rooms/index.html.erb

```
<h1>Chat rooms</h1>

<p class="lead"><%= link_to 'New chat room', new_chat_room_path, class: 'btn btn-primary' %></p>

<ul>
  <%= render @chat_rooms %>
</ul>
```

views/chat_rooms/_chat_room.html.erb

```
<li><%= link_to "Enter #{chat_room.title}", chat_room_path(chat_room) %></li>
```

views/chat_rooms/new.html.erb

```
<h1>Add chat room</h1>

<%= form_for @chat_room do |f| %>
  <div class="form-group">
    <%= f.label :title %>
    <%= f.text_field :title, autofocus: true, class: 'form-control' %>
  </div>

  <%= f.submit "Add!", class: 'btn btn-primary' %>
<% end %>
```

Messages

The main star of our app is, of course, a chat message. It should belong to both a user and a chat room. To get there, run the following:

```
$ rails g model Message body:text user:references chat_room:references
$ rails db:migrate
```

Make sure to establish the proper relations:

models/chat_room.rb

```
[...]
belongs_to :user
has_many :messages, dependent: :destroy
[...]
```

models/users.rb

```
[...]
has_many :chat_rooms, dependent: :destroy
has_many :messages, dependent: :destroy
[...]
```

models/message.rb

```
[...]
belongs_to :user
belongs_to :chat_room
[...]
```

So far, so good. Messages should be displayed when a user enters a chat room, so create a new `show` action:

chat_rooms_controller.rb

```
[...]
def show
  @chat_room = ChatRoom.includes(:messages).find_by(id: params[:id])
end
[...]
```

Note the includes (http://api.rubyonrails.org/classes/ActiveRecord/QueryMethods.html#method-i-includes) method here used for eager loading.

Now the views:

views/chat_rooms/show.html.erb

```
<h1><%= @chat_room.title %></h1>

<div id="messages">
  <%= render @chat_room.messages %>
</div>
```

views/messages/_message.html.erb

```
<div class="card">
  <div class="card-block">
    <div class="row">
      <div class="col-md-1">
        <%= gravatar_for message.user %>
      </div>
      <div class="col-md-11">
        <p class="card-text">
          <span class="text-muted"><%= message.user.name %> at <%= message.timestamp %> says</span><br>
          <%= message.body %>
        </p>
      </div>
    </div>
  </div>
</div>
```

In this partial three new methods are employed: `user.name`, `message.timestamp` and `gravatar_for`. To construct a name, let's simply strip off the domain part from the user's email (of course, in a real app you'd want to allow them entering a name upon registration or at the "Edit profile" page):

models/user.rb

```
[...]
def name
  email.split('@')[0]
end
[...]
```

`timestamp` relies on `strftime` to present message's creation date in a user-friendly format:

models/message.rb

```
[...]
def timestamp
  created_at.strftime('%H:%M:%S %d %B %Y')
end
[...]
```

`gravatar_for` is a helper to display user's gravatar:

application_helper.rb

```
    ()
module ApplicationHelper
  def gravatar_for(user, opts = {})
    opts[:alt] = user.name
    image_tag "https://www.gravatar.com/avatar/#{Digest::MD5.hexdigest(user.email)}?s=#{opts.delete(:size) { 40
}}",
              opts
  end
end
```

The last two things to do here is to style the messages container a bit:

```
#messages {
  max-height: 450px;
  overflow-y: auto;
  .avatar {
    margin: 0.5rem;
  }
}
```

Add routes:

config/routes.rb

```
[...]
resources :chat_rooms, only: [:new, :create, :show, :index]
root 'chat_rooms#index'
[...]
```

Finally, preparations are done and we can proceed to coding the core functionality of our chat.

Adding ActionCable

Client Side

Before proceeding, install Redis on your machine if you do not already have it. Redis is available for nix (http://redis.io/download), via Homebrew (http://brew.io) and for Windows (https://github.com/MSOpenTech/redis), as well.

Next, tweak the Gemfile:

Gemfile

```
[...]
gem 'redis', '~> 3.2'
[...]
```

and run

```
$ bundle install
```

Now you may modify the *config/cable.yml* file to use Redis as an adapter:

config/cable.yml

```
[...]
adapter: redis
url: YOUR_URL
[...]
```

Or simply use `adapter: async` (the default value).

Also, modify your *routes.rb* to mount ActionCable on some URL:

config/routes.rb

```
[...]
mount ActionCable.server => '/cable'
[...]
```

Check that inside the *javascripts* directory there is a *cable.js* file with the contents like:

javascripts/cable.js

```
//= require action_cable
//= require_self
//= require_tree ./channels

(function() {
  this.App || (this.App = {});

  App.cable = ActionCable.createConsumer();

}).call(this);
```

This file must be required inside *application.js*:

javascripts/application.js

```
[...]
//= require cable
[...]
```

`Consumer` is a client of a web socket connection that can subscribe to one or multiple channels. Each ActionCable server may handle multiple connections. `Channel` is similar to an MVC controller and is used for streaming. You may read more about ActionCable's terminology here (https://github.com/rails/rails/tree/master/actioncable#terminology).

So, let's create a new channel:

javascripts/channels/rooms.coffee

```
(()
App.global_chat = App.cable.subscriptions.create {
    channel: "ChatRoomsChannel"
    chat_room_id: ''
  },
  connected: ->
    # Called when the subscription is ready for use on the server

  disconnected: ->
    # Called when the subscription has been terminated by the server

  received: (data) ->
    # Data received

  send_message: (message, chat_room_id) ->
    @perform 'send_message', message: message, chat_room_id: chat_room_id
```

Here, we basically subscribe a consumer to the `ChatRoomsChannel` and pass the current room's id (at this point we do not really pass anything, but that'll be fixed soon). The subscription has a number of self-explaining callbacks: `connected`, `disconnected`, and `received`. Also, the subscription defines the main function (`send_message`) that invokes the method with the same name of the server and passes the necessary data to it.

Of course, we need a form to allow users to send their messages:

views/chat_rooms/show.html.erb

```
<%= form_for @message, url: '#' do |f| %>
  <div class="form-group">
    <%= f.label :body %>
    <%= f.text_area :body, class: 'form-control' %>
    <small class="text-muted">From 2 to 1000 characters</small>
  </div>

  <%= f.submit "Post", class: 'btn btn-primary btn-lg' %>
<% end %>
```

The `@message` instance variable should be set inside the controller:

chat_rooms_controller.rb

```
[...]
def show
  @chat_room = ChatRoom.includes(:messages).find_by(id: params[:id])
  @message = Message.new
end
[...]
```

Of course, you might use the basic `form` tag instead of relying on the Rails form builder, but this allows us to take advantage of things like I18n translations later.

Let's also add some validations for messages:

models/message.rb

```
    ()
[...]
validates :body, presence: true, length: {minimum: 2, maximum: 1000}
[...]
```

Another problem to tackle here is providing our script with the room's id. Let's solve it with the help of HTML `data-` attribute:

views/chat_rooms/show.html.erb

```
[...]
<div id="messages" data-chat-room-id="<%= @chat_room.id %>">
  <%= render @chat_room.messages %>
</div>
[...]
```

Having this in place, we can use room's id in the script:

javascripts/channels/rooms.coffee

```
jQuery(document).on 'turbolinks:load', ->
  messages = $('#messages')
  if $('#messages').length > 0

    App.global_chat = App.cable.subscriptions.create {
        channel: "ChatRoomsChannel"
        chat_room_id: messages.data('chat-room-id')
      },
      connected: ->
        # Called when the subscription is ready for use on the server

      disconnected: ->
        # Called when the subscription has been terminated by the server

      received: (data) ->
        # Data received

      send_message: (message, chat_room_id) ->
        @perform 'send_message', message: message, chat_room_id: chat_room_id
```

Note the `jQuery(document).on 'turbolinks:load'` part. This should be done only if you are using Turbolinks (https://github.com/turbolinks/turbolinks-classic) 5 that supports this new event. You might think about usng jquery-turbolinks (https://rubygems.org/gems/jquery-turbolinks) to bring the default jQuery events back, but unfortunately it is not compatible with Turbolinks 5 (https://github.com/kossnocorp/jquery.turbolinks/issues/56).

The logic of the script is pretty simple: check if there is a `#messages` block on the page and, if yes, subscribe to the channel while providing the room's id. The next step is to listen for the form's `submit` event:

javascripts/channels/rooms.coffee

```
(/)
jQuery(document).on 'turbolinks:load', ->
  messages = $('#messages')
  if $('#messages').length > 0

    App.global_chat = App.cable.subscriptions.create
    # ...

    $('#new_message').submit (e) ->
      $this = $(this)
      textarea = $this.find('#message_body')
      if $.trim(textarea.val()).length > 1
        App.global_chat.send_message textarea.val(), messages.data('chat-room-id')
        textarea.val('')
      e.preventDefault()
      return false
```

When the form is submitted, take the message's body, check that its length is at least two and then call the `send_message` function to broadcast the new message to all visitors of the chat room. Next, clear the textarea and prevent form submission.

Server Side

Our next task will be to introduce a channel on our server. In Rails 5, there is a new directory called *channels* to host them, so create a *chat_rooms_channel.rb* file there:

channels/chat_rooms_channel.rb

```
class ChatRoomsChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_rooms_#{params['chat_room_id']}_channel"
  end

  def unsubscribed
    # Any cleanup needed when channel is unsubscribed
  end

  def send_message(data)
    # process data sent from the page
  end
end
```

`subscribed` is a special method to start streaming from a channel with a given name. As long as we have multiple rooms, the channel's name will vary. Remember, we provided `chat_room_id: messages.data('chat-room-id')` when subscribing to a channel in our script? Thanks to it, `chat_room_id` can be fetched inside the `subscribed` method by calling `params['chat_room_id']`.

`unsubscribed` is a callback that fires when the streaming is stopped, but we won't use it in this demo.

The last method – `send_message` – fires when we run `@perform 'send_message', message: message, chat_room_id: chat_room_id` from our script. The `data` variable contains a hash of sent data, so, for example, to access the message you would type `data['message']`.

There are multiple ways to broadcast the received message, but I am going to show you a very neat solution based on the demo provided by DHH (I've also found this article (http://www.akitaonrails.com/2015/12/28/fixing-dhh-s-rails-5-chat-demo) with a slightly different approach).

First of all, modify the `send_message` method:

channels/chat_rooms_channel.rb

```
[...]
def send_message(data)
  current_user.messages.create!(body: data['message'], chat_room_id: data['chat_room_id'])
end
[...]
```

Once we receive a message, save it to the database. You don't even need to check whether the provided chat room exists – by default, in Rails 5, a record's parent must exist in order to save it. This behavior can be changed by setting `optional: true` for the `belongs_to` relation (read about other changes in Rails 5 here (https://www.sitepoint.com/onwards-to-rails-5-additions-changes-and-deprecations/)).

There is a problem though – Devise's `current_user` method is not available for us here. To fix that, modify the *connection.rb* file inside the *application_cable* directory:

channels/application_cable/connection.rb

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
      logger.add_tags 'ActionCable', current_user.email
    end

    protected

    def find_verified_user # this checks whether a user is authenticated with devise
      if verified_user = env['warden'].user
        verified_user
      else
        reject_unauthorized_connection
      end
    end
  end
end
```

Having this in place, we achieve even two goals at once: the `current_user` method is now available for the channel and unauthenticated users are not able to broadcast their messages.

The call to `logger.add_tags 'ActionCable', current_user.email` is used to display debugging information in the console, so you will see output similar to this:

```
[ActionCable] [test@example.com] Registered connection (Z2lkOi8vY2FibGUtY2hhdC9Vc2VyLzE)
[ActionCable] [test@example.com] ChatRoomsChannel is transmitting the subscription confirmation
[ActionCable] [test@example.com] ChatRoomsChannel is streaming from chat_rooms_1_channel
```

Under the hood Devise uses Warden (https://github.com/hassox/warden) for authentication, so `env['warden'].user` tries to fetch the currently logged-in user. If it is not found, `reject_unauthorized_connection` forbids broadcasting.

Now, let's add a callback that fires after the message is actually saved to the database to schedule a background job:

models/message.rb

```
[...]
after_create_commit { MessageBroadcastJob.perform_later(self) }
[...]
```

In this callback `self` is a saved message, so we basically pass it to the job. Write the job now:

jobs/message_broadcast_job.rb

```
class MessageBroadcastJob < ApplicationJob
  queue_as :default

  def perform(message)
    ActionCable.server.broadcast "chat_rooms_#{message.chat_room.id}_channel",
                                 message: 'MESSAGE_HTML'
  end
end
```

The `perform` method does the actual broadcasting, but what about the data we want to broadcast? Once again, there are a couple of ways to solve this problem. You may send JSON with the message data and then on the client side use a templating engine like Handlebars (http://handlebarsjs.com/). In this demo, however, let's send HTML markup from the *messages/_message.html.erb* partial we created earlier. This partial can be rendered with the help of a controller:

jobs/message_broadcast_job.rb

```
class MessageBroadcastJob < ApplicationJob
  queue_as :default

  def perform(message)
    ActionCable.server.broadcast "chat_rooms_#{message.chat_room.id}_channel",
                                 message: render_message(message)
  end

  private

  def render_message(message)
    MessagesController.render partial: 'messages/message', locals: {message: message}
  end
end
```

In order for this to work, you'll have to create an empty `MessagesController`:

messages_controller.rb

```
class MessagesController < ApplicationController
end
```

Back to the Client Side

Great, now the server side is ready and we can finalize our script. As long as we broadcast HTML markup, it can be simply placed right onto the page without any further manipulations:

javascripts/channels/rooms.coffee

```
[...]
App.global_chat = App.cable.subscriptions.create {
    channel: "ChatRoomsChannel"
    chat_room_id: messages.data('chat-room-id')
  },
  connected: ->
    # Called when the subscription is ready for use on the server

  disconnected: ->
    # Called when the subscription has been terminated by the server

  received: (data) ->
    messages.append data['message']

  send_message: (message, chat_room_id) ->
    @perform 'send_message', message: message, chat_room_id: chat_room_id
[...]
```

The only thing I don't really like here is that, by default, the user sees old messages, whereas the newer ones are being placed at the bottom. You could either use the `order` method to sort them properly and replace `append` with `prepend` inside the `received` callback, but I'd like to make our chat behave like Slack. In Slack, newer messages are also placed at the bottom, but the chat window automatically scrolls to them. That's easy to achieve with the following function that is called once the page is loaded:

javascripts/channels/rooms.coffee

```
jQuery(document).on 'turbolinks:load', ->
  messages = $('#messages')
  if $('#messages').length > 0
    messages_to_bottom = -> messages.scrollTop(messages.prop("scrollHeight"))

    messages_to_bottom()

    App.global_chat = App.cable.subscriptions.create
    # ...
```

Let's also scroll to the bottom once a new message has arrived (because by default it won't be focused):

javascripts/channels/rooms.coffee

```
[...]
received: (data) ->
  messages.append data['message']
  messages_to_bottom()
[...]
```

Great! Check out the resulting script on GitHub (https://github.com/bodrovis/Sitepoint-source/blob/master/Chat_with_ActionCable_and_Devise/app/assets/javascripts/channels/rooms.coffee).

Pushing to Heroku

If you wish to push your new shiny chat to Heroku, some additional actions have to be taken. First of all, you will have install a Redis addon. There are many addons (https://elements.heroku.com/) to choose from: for example, you could use Rediscloud (https://elements.heroku.com/addons/rediscloud). When the addon is installed, tweak *cable.yml* to provide the proper Redis URL. For Rediscloud it is stored inside the `ENV["REDISCLOUD_URL"]` environment variable:

config/cable.yml

```
production:
  adapter: redis
  url: <%= ENV["REDISCLOUD_URL"] %>
[...]
```

The next step is to list the allowed origins to subscribe to the channels:

config/environments/production.rb

```
[...]
config.action_cable.allowed_request_origins = ['https://your_app.herokuapp.com',
                                               'http://your_app.herokuapp.com']
[...]
```

Lastly, you have to provide the ActionCable URL. As long as our *routes.rb* has `mount ActionCable.server => '/cable'`, the corresponding setting should be:

config/environments/production.rb

```
[...]
config.action_cable.url = "wss://sitepoint-actioncable.herokuapp.com/cable"
[...]
```

Having this in place, you can push your code to Heroku and observe the result. Yay!

Conclusion

In this article we've discussed how to set up ActionCable and code a mini-chat app with support for multiple rooms. The app includes both the client and server sides, while providing a basic authorization mechanism. We also employed a background job to power up the broadcasting process and discussed steps required to deploy the application to Heroku.

Hopefully, you found this article useful and interesting. Have you already tried using ActionCable? Did you like it? Share your opinion in the comments. Follow me on Twitter to be the first one to know about my articles, and see you soon!

Learn more on ruby with our tutorial Simulate User Behavior and Test Your Ruby App (%E2%80%9Chttps://www.sitepoint.com/premium/screencasts/feature-tests-with-rspec-simulate-user-behavior-and-test-your-ruby-app%E2%80%9D) on SitePoint.


(htt

Meet the author

Ilya Bodrov-Krukowski (https://www.sitepoint.com/author/ibodrov/) (https://twitter.com/bodrovis) (https://plus.google.com/103641984440210150447) (https://facebook.com/isbodrov) (https://linkedin.com/in/bodrovis) (https://github.com/bodrovis)

Ilya Bodrov is personal IT teacher, a senior engineer working at Campaigner LLC, author and teaching assistant at Sitepoint and lecturer at Moscow Aviations Institute. His primary programming languages are Ruby (with Rails) and JavaScript. He enjoys coding, teaching people and learning new things. Ilya also has

some Cisco and Microsoft certificates and was working as a tutor in an educational center for a couple of years. In his free time he tweets, writes posts for his website (http://bodrovis.tech), participates in OpenSource projects, goes in for sports and plays music.