AkitaOnRails

Fixing DHH's Rails 5 Chat Demo

2015 December 28, 16:11 h - tags: pusher rails5 actioncable websocket

So, Rails 5.0.0 Beta 1 has just been released and the main new feature is Action Cable.

It's basically a complete solution on top of vanilla Rails so you can implement WebSocket based applications (the usual real time chats and notifications) with full access to your Rails assets (models, view templates, etc). For small to medium apps, this is a terrific solution that you might want to use instead of having to go to Node.js.

In summary you control Cable Channels that can receive messages sent through a WebSocket client wiring. The new Channel generator takes care of the boilerplate and you just have to fill in the blanks for what kinds of messages you want to send from the client, what you want to broadcast from the server, and to what channels that your clients are subscribed to.

For a more in-depth introduction, DHH himself published a bare bone Action Cable screencast that you should watch just to get a feeling of what the fuzz is all about. If you watched it already and have experience in programming, you may have spotted the problem I mention in the title, so just jump to "The Problem" section below for a TL;DR.

In the end you will end up with a code base like the one I reproduced in my Github repository up until the tag "end_of_dhh". You will have a (very) bare bone single-room real time chat app for you to play with the main components.

Let's just list the main components here. First, you will have the ActionCable server mounted in the "routes.rb" file:

```
# config/routes.rb
```

```ruby
Rails.application.routes.draw do
  root to: 'rooms#show'

  # Serve websocket cable requests in-process
  mount ActionCable.server => '/cable'
end
```

This is the main server component, the channel:

```ruby
# app/channels/room_channel.rb
class RoomChannel < ApplicationCable::Channel
  def subscribed
    stream_from "room_channel"
  end

  def unsubscribed
    # Any cleanup needed when channel is unsubscribed
  end

  def speak(data)
    Message.create! content: data['message']
  end
end
```

Then you have the boilerplate Javascript:

```coffeescript
# app/assets/javascripts/cable.coffee
#= require action_cable
#= require_self
#= require_tree ./channels
#
@App ||= {}
App.cable = ActionCable.createConsumer()
```

And the main client-side Websocket hooks:

```coffeescript
# app/assets/javascripts/channels/room.coffee
App.room = App.cable.subscriptions.create "RoomChannel",
  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']

  speak: (message) ->
    @perform 'speak', message: message

$(document).on "keypress", "[data-behavior~=room_speaker]", (event) ->
  if event.keyCode is 13
    App.room.speak event.target.value
    event.target.value = ''
    event.preventDefault()
```

The view template is a bare bone HTML just to hook a simple form and div to list the messages:

```
<!-- app/views/rooms/show.html.erb -->
<h1>Chat room</h1>

<div id="messages">
  <%= render @messages %>
</div>

<form>
  <label>Say something:</label><br>
  <input type="text" data-behavior="room_speaker">
</form>
```

The Problem

In the "RoomChannel", you have the "`speak`" method that saves a message to the database. This is already a red flag for a WebSocket action that is supposed to have very short lived, light processing. Saving to the database is to be considered heavyweight, specially under load. If this is processed inside EventMachine's reactor loop, it will block the loop and avoid other concurrent processing to take place until the database releases the lock.

```ruby
# app/channels/room_channel.rb
class RoomChannel < ApplicationCable::Channel
  ...
  def speak(data)
    Message.create! content: data['message']
  end
end
```

I would say that anything that goes inside the channel should be asynchronous!

To add harm to injury, this is what you have in the "Message" model itself:

```ruby
class Message < ApplicationRecord
  after_create_commit { MessageBroadcastJob.perform_later self }
end
```

A model callback (avoid those as the plague!!) to broadcast the received messsage to the subscribed Websocket clients as an ActiveJob that looks like this:

```ruby
class MessageBroadcastJob < ApplicationJob
  queue_as :default

  def perform(message)
    ActionCable.server.broadcast  'room_channel', message: render_message(messa
  end

  private

  def render_message(message)
    ApplicationController.renderer.render(partial: 'messages/message', locals:
  end
end
```

It renders the HTML snippet to send back for the Websocket clients to append to their browser DOMs.

DHH even goes on to say *"I'd like to show it because this is how most apps will end up."*

Indeed, the **problem** is that most people will just follow this pattern and it's a big trap. So, what's the solution instead?

The Proper Solution

For just the purposes of a simple screencast, let's make a quick fix.

First of all, if at all possible you want your channel code to block as little as possible. Waiting for a blocking operation in the database (writing) is definitely not one of them. The Job is underused, it should be called straight from the channel "speak" method, like this:

```
# app/channels/room_channel.rb
 class RoomChannel < ApplicationCable::Channel
   ...
   def speak(data)
-    Message.create! content: data['message']
+    MessageBroadcastJob.perform_later data['message']
   end
 end
```

Then, we move the model writing to the Job itself:

```
# app/jobs/message_broadcast_job.rb
 class MessageBroadcastJob < ApplicationJob
   queue_as :default

-  def perform(message)
-    ActionCable.server.broadcast  'room_channel', message: render_message(mess
+  def perform(data)
+    message = Message.create! content: data
+    ActionCable.server.broadcast 'room_channel', message: render_message(messa
   end
   ...
```

And finally, we remove that horrible callback from the model and make it bare-bone again:

```ruby
# app/models/message.rb
class Message < ApplicationRecord
end
```

This returns quickly, defer processing to a background job and should sustain more concurrency out-of-the-box. The previous, DHH solution, have a built-in bottleneck in the speak method and will choke as soon as the database becomes the bottleneck.

It's by no means a perfect solution yet, but it's less terrible for a very quick demo and the code ends up being simpler as well. You can check out this code in my Github repo commit.

I may be wrong in the conclusion that the channel will block or if this is indeed harmful for the concurrency. I didn't measure both solutions, it's just a gut feeling from older wounds. If you have more insight into the implementation of Action Cable, leave a comment down below.

By the way, be careful before considering migrating your Rails 4.2 app to Rails 5 just yet. Because of the hard coded dependencies on Faye, Eventmachine, Rails 5 right now rules out Unicorn (even Thin seems to be having problem booting up). It also rules out JRuby and MRI on Windows as well because of Eventmachine.

If you want the capabilities of Action Cable without having to migrate, you can use solutions such as "Pusher.com", or if you want your own in-house solution, follow my evolution on the subject with my mini-Pusher clone written in Elixir.

Comments

4 Comentários **AkitaOnRails** **Milan Rawal**

Recomendar 3 Compartilhar Ordenar por Mais recentes

Participe da discussão...

Nico • um ano atrás

Why are you so much against model callbacks? Yes, they can be tricky to deal with at times, but they are also quite helpful many times...

• Responder • Compartilhar ›

yan • um ano atrás

THANK YOU SO MUCH!!!! This totally fixed the problem I was having! Your a genius! te amo amigo!

• Responder • Compartilhar ›

Tiago Cardoso • 2 anos atrás

Are you sure that the speak is/was handled within the EM loop? As far as I understood ActionCable from what I read, the messages come from the loop, but are passed to a separate thread pool to be handled (beta1 was celluloid actors, beta2 was a thread executor pool from concurrent-ruby, if I'm not mistaken). You are indeed bound by the blocking operations (communicate with DB will block the actor/thread), but you will not block the EM loop.

• Responder • Compartilhar ›

Rodrigo Rosenfeld Rosas • 2 anos atrás

Honestly, I don't see any big flaws in DHH's demo. He possibly extracted part of what they have implemented for Basecamp. I don't trust the current implementation of ActionCable to be really thread-safe and would be worried to use it for some purposes, but for simple cases and most applications, which don't demand a really high throughput for websockets, I think his approach was fine enough.

Creating a simple record in Sequel/PostgreSQL is pretty pretty fast (very often less than 1ms), so I don't think it would make much different for most applications to move that creation to the job.

On the other side, we could assume that the message could have been generated by some event other than a websocket request. Now, what if a message is created as part of some transaction. Forgive me to give some Sequel example as there's a long time since I touched AR and I don't remember its syntax for transactions:

DB.transaction do
Message.create! message: message
save_some_file_on_disk
end

Through the after_commit hook, DHH's solution should work fine regardless of the transaction succeeding or not in case there's a problem while saving the file to disk. Your solution could be valid but it's hard to evaluate the best option without understanding the use cases for the application and understanding how much would be its throughtput and performing some benchmarks...

4 • Responder • Compartilhar ›

TAMBÉM EM **AKITAONRAILS**

Ex Manga Downloadr - Part 7: Properly dealing with large Collections | …

1 comentário • 4 meses atrás

AvatarMuhammad Muhajir — I have some of your

Coherence and ExAdmin - Devise and ActiveAdmin for Phoenix | …

11 comentários • 10 meses atrás

AvatarAkitaOnRails — Don't worry. Super scalable.

AvatarMuhammad Mahaj... — I have some of your posts, do you elixir over crystal?

AvatarAkitaOnRails — Don't worry. Super scalable, Super high concurrency are important, but for the 1%. For the 99% of the Web, Rails is still King.

The Economics of Software Development | AkitaOnRails.com

3 comentários • 4 meses atrás

AvatarAkitaOnRails — You "can" in basically any methodology, including Waterfall for that matter. 99% of people using Kanban use it just as any …

Upcoming built-in upload Solution for Rails 5.2 (ActiveStorage) |

13 comentários • 3 meses atrás

AvatarNguyen Duc Phuong — Okay, i was asking because I have no problem accepting direct large file or slow client upload with Heroku.