

Show nav

[Heroku](#)

- [Products](#)
  - [Heroku Platform](#)
    - [Heroku DX](#)
      - [Heroku Flow](#)
      - [Continuous Delivery](#)
      - [Continuous Integration](#)
    - [Heroku OpEx](#)
    - [Heroku Runtime](#)
      - [Heroku Dynos](#)
  - [Heroku Postgres](#)
  - [Heroku Redis](#)
  - [Kafka on Heroku](#)
  - [Heroku Connect](#)
  - [Heroku Enterprise](#)
    - [Private Spaces](#)
  - [Heroku Teams](#)
- [Elements](#)
  - [Add-ons](#)
  - [Buttons](#)
  - [Buildpacks](#)
  - [About](#)
- [Pricing](#)
- [Documentation](#)
- [Support](#)
- [More](#)

Resources

- [What is Heroku?](#)
- [Help](#)
- [Customers](#)
- [Careers](#)
- [Events](#)

Heroku is for

- [Developers](#)
- [CTOs](#)
- [Team Collaboration](#)
- [Startups](#)
- [Enterprises](#)
- [Agencies](#)

Languages

- [Node](#)
- [Ruby](#)
- [Java](#)
- [PHP](#)
- [Python](#)
- [Go](#)
- [Scala](#)
- [Clojure](#)

Latest News from the Heroku Blog

## [Container Registry & Runtime GA: Deploy Docker Images to Heroku](#)

8 days ago by Jon Byrum

In the last few years Docker has emerged as a de facto standard for packaging apps for deployment. Today, Heroku [Read More](#)

[More news](#) [View all blog posts](#)

- 
- [Log in](#) or [Sign up](#)

## [Heroku Blog](#)

- [Latest](#)
- [News](#)
- [Engineering](#)
- [Blog](#) /
- [News](#) /
- Real-Time Rails: Implementing WebSockets in Rails 5 with Action Cable

## Real-Time Rails: Implementing WebSockets in Rails 5 with Action Cable



May 09, 2016 . Posted by [Sophie DeBenedetto](#)

---

*It's been one year since Action Cable debuted at RailsConf 2015, and Sophie DeBenedetto is here to answer the question in the minds of many developers: what is it really like to implement "the highlight of Rails 5"? [Sophie](#) is a web developer and an instructor at the Flatiron School. Her first love is Ruby on Rails, although she has developed projects with and written about Rails, Ember and Phoenix.*

---

Recent years have seen the rise of "the real-time web." Web apps we use every day rely on real-time features—the sort of features that let you see new posts magically appearing at the top of your feeds without having to lift a finger.

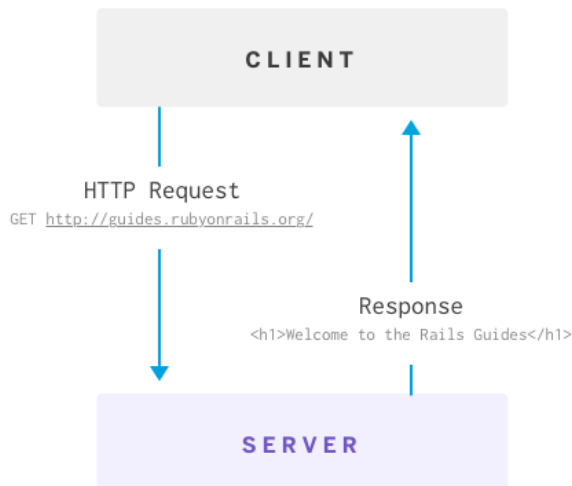
While we may take those features for granted, they represent a significant departure from the HTTP protocol's strict request-response pattern. Real-time web, by contrast, loosely describes a system in which users receive new information from the server as soon as it is available—no request required.

There are a number of strategies and technologies for implementing such real-time functionality, but the [WebSocket protocol](#) has been rising to prominence since its development in 2009. However, up until very recently, implementing the WebSocket protocol in Rails was difficult. There was no native support, and any real-time feature required integrating third party libraries and strategies like Faye or JavaScript polling. So let's take a closer look at WebSockets and how Rails 5 has evolved to support real-time apps with Action Cable.

### [What are WebSockets?](#)

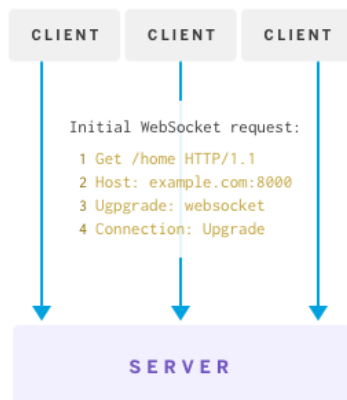
WebSockets are a protocol built on top of TCP. They hold the connection to the server open so that the server can send information to the client, even in the absence of a request from the client. WebSockets allow for bi-directional, "full-duplex" communication between the client and the server by creating a persistent connection between the two.

## HTTP PROTOCOL

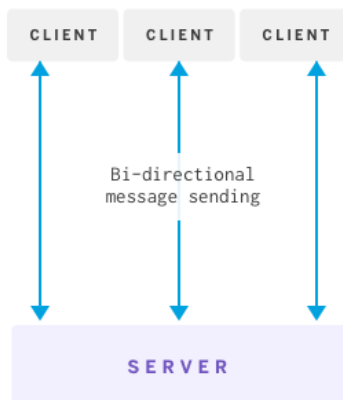


## WEBSOCKET PROTOCOL

**Step 1:** A Request to initiate a WebSocket connection is sent to the server, from any number of clients



**Step 2:** Open and maintain WebSocket connections between multiple clients and a single server



With the development of Action Cable and its recent integration into Rails 5, we now have a full-stack, easy-to-use implementation of WebSockets that follows the Rails design patterns we've come to rely on. The only question is why it took so long.

## The Path to Real-Time Rails

In 2015, Rails' [benevolent-dictator-for-life DHH](#) changed his tune about sockets. He started by acknowledging an important truth: that ["dealing with WebSockets is a pain in the \[you know what\]."](#) And although it wasn't necessarily a pleasure to code, you *could* build real-time features into Rails with nothing more than Faye and Javascript polling. In fact, Campfire, Basecamp's own chatting application, has been using polling for about a decade, and I've built compelling real-time features that way too.

But DHH knew there's something lost without sockets. "If you can make WebSockets even less work than polling, why wouldn't you do it?" Sure, polling met the needs of his team (and many others) for many years. But as more and more consumers and developers began demanding real-time functionality, and as newer frameworks like [Phoenix](#) arrived to meet that demand, Rails felt the need to deliver—and in fact, Action Cable draws some inspiration from Phoenix channels.

It hasn't been smooth sailing. I've followed the development of Action Cable closely, and before it was merged into Rails 5, I would say that it *wasn't* easier than polling. But after a year of development, it's very easy to implement, and it aligns nicely with the other design patterns we've become so comfortable with in Rails.

So, how does the "highlight" of Rails 5 work, and what's it like to implement? Let's take a closer look!

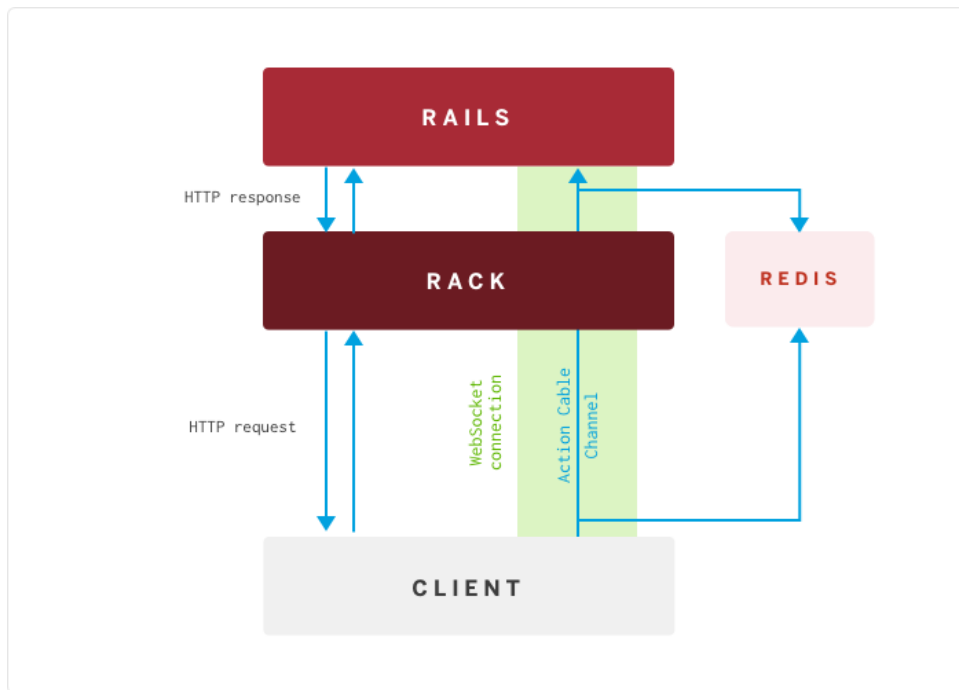
## Introducing Action Cable

So what do we have to look forward to? Well, it's what [the docs](#) call a "full-stack offering": it provides both a client-side JavaScript framework, and a server-side Ruby framework. And because it integrates so tightly with Rails, we have access to all of our models from within our WebSocket workers, effectively layering Action Cable on top of our existing Rails architecture, including Active Record (or any other ORM).

### Action Cable Under the Hood

Before we dive into some code, let's take a closer look at how Action Cable opens and maintains the WebSocket connection inside our Rails 5 application.

Action Cable can be run on a stand-alone server, or we can configure it to run on its own processes within the main application server. In this post, we'll be taking a look at the second approach.



Action Cable uses the Rack socket hijacking API to take over control of connections from the application server. Action Cable then manages connections internally, in a multithreaded manner, layering as many channels as you care to define over that socket connection.

For every instance of your application that spins up, an instance of Action Cable is created, using Rack to open and maintain a persistent connection, and using a channel mounted on a sub-URI of your main application to stream from certain areas of your application and broadcast to other areas.

Action Cable offers server-side code to broadcast certain content (think new messages or notifications) over the channel, to a subscriber. The subscriber is instantiated on the client-side with a handy JavaScript function that uses jQuery to append new content to the DOM.

Lastly, Action Cable uses Redis as a data store for transient data, syncing content across instances of your application.

Now that we have a basic understanding of how Action Cable works, we'll build out a basic chatting application in Rails 5, taking a closer look at how Action Cable behaves along the way.

## Building a Real-Time Chat App with Action Cable

### Getting Started: Application Architecture

In this example, we'll build a basic chatting application in which users can log in or sign up to create a username, then create a new chat room or choose from an existing chat room and start messaging. We'll use Action Cable to ensure that our chat feature is real-time: any users in a given chat room will see new messages (their own and the messages of other users) appear in the chat, without reloading the page or engaging in any other action to request new content.

You can follow along below, check out the code for this project on [GitHub](#), visit my deployment [here](#) or, deploy your own copy by clicking this button.



## Starting a new Rails 5 app with Action Cable

At the time of this writing, Rails 5 was in Beta 3. So, to start a new Rails 5 app, we need to do the following.

- First, make sure you have installed and are using Ruby 2.3.0.
- Then:

```
$ gem install rails --pre
Successfully installed rails-5.0.0.beta3
Parsing documentation for rails-5.0.0.beta3
Done installing documentation for rails after 1 seconds
1 gem installed
```

- You'll also need to have Postgres and Redis installed in order to build this app. You can `brew install redis` and `brew install postgres` if you haven't done so already.
- Now we're ready to generate our new app!

```
rails new action-cable-example --database=postgresql
```

This will generate a shiny new Rails 5 app, complete with all kinds of goodies in our Gemfile. Open up the Gemfile and you'll notice that we have:

- The latest version of Rails

```
# Gemfile
gem 'rails', '>= 5.0.0.beta3', '< 5.1'
```

- Redis (which we will need to un-comment out):

```
# Gemfile
# gem 'redis', '~> 3.0'
```

- Puma (since Action Cable needs a threaded server):

```
# Gemfile
gem 'puma'
```

Make sure you've un-commented out the Redis gem from your Gemfile, and `bundle install`.

## Domain Model

Our domain model is simple: we have users, chat rooms and messages.

A **chat room** will have a topic and it will have many messages. A **message** will have content, and it will belong to a user and belong to a chatroom. A **user** will have a username, and will have many messages.

The remainder of this tutorial will assume that we have already generated the migrations necessary to create the chat rooms, messages and users table, and that we have already defined our Chatroom, User and Message model to have the relationships described above. Let's take a quick look at our models before we move on.

```
# app/models/chatroom.rb

class Chatroom < ApplicationRecord
  has_many :messages, dependent: :destroy
  has_many :users, through: :messages
  validates :topic, presence: true, uniqueness: true, case_sensitive: false
end

# app/models/message.rb

class Message < ApplicationRecord
  belongs_to :chatroom
  belongs_to :user
end

# app/models/user.rb

class User < ApplicationRecord
  has_many :messages
  has_many :chatrooms, through: :messages
  validates :username, presence: true, uniqueness: true
end
```

## Application Flow

We'll also assume that our routes and controllers are up and running. A user can log in with a username, visit a chat room and create new messages via a form on the chat room's show page.

Accordingly, the #show action of the Chatrooms Controller sets up both a @chatroom instance as well as a new, empty @message instance that will be used to build our form for a new message, on the chatroom's show page:

```
# app/controllers/chatrooms_controller.rb

class ChatroomsController < ApplicationController
  ...

  def show
    @chatroom = Chatroom.find_by(slug: params[:slug])
    @message = Message.new
  end
end
```

Our chatroom show page renders the partial for the message form:

```
# app/views/chatrooms/show.html.erb

<div class="row col-md-8 col-md-offset-2">
  <h1><%= @chatroom.topic %></h1>

  <div class="panel panel-default">
    <% if @chatroom.messages.any? %>
      <div class="panel-body" id="messages">
        <%= render partial: 'messages/message', collection: @chatroom.messages%>
      </div>
    <%else%>
      <div class="panel-body hidden" id="messages">
      </div>
    <%end%>
  </div>

  <%= render partial: 'messages/message_form', locals: {message: @message, chatroom: @chatroom}%>
</div>
```

And our form for a new message looks like this:

```
# app/views/messages/_message_form.html.erb

<%= form_for message, remote: true, authenticity_token: true do |f| %>
  <%= f.label :your_message %>
  <%= f.text_area :content, class: "form-control", data: {textarea: "message"} %>

  <%= f.hidden_field :chatroom_id, value: chatroom.id %>
  <%= f.submit "send", class: "btn btn-primary", style: "display: none", data: {send: "message"} %>
<% end %>
```

Our new message form posts to the create action of the Messages Controller

```
class MessagesController < ApplicationController

  def create
    message = Message.new(message_params)
    message.user = current_user
    if message.save
      # do some stuff
    else
      redirect_to chatrooms_path
    end
  end

  private

  def message_params
    params.require(:message).permit(:content, :chatroom_id)
  end
end
```

Okay, now that we have our app up and running, let's get those real-time messages working with Action Cable.

## Implementing Action Cable

Before we define our very own Messages Channel and start working directly with Action Cable code, let's take a quick tour of the Action Cable infrastructure that Rails 5 provides for us.

When we generated our brand new Rails 5 application, the following directory was generated for us:

```
├── app
│   └── channels
│       ├── application_cable
│       ├── channel.rb
│       └── connection.rb
```

Our ApplicationController module has a Channel and a Connection class defined for us.

The `Connection` class is where we would authorize the incoming connection—for example, establishing a channel to a given user's inbox, which requires user authorization. We'll leave this class alone, as any user can join any chat room at any time. However, the `Messages Channel` that we will define shortly will inherit from `ApplicationCable::Channel`.

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    end
end
```

The `Channel` class is where we would place shared logic among any additional channels that we will define. We'll only be creating one channel, the `Messages Channel`, so we'll leave this class alone as well.

```
# app/channels/channel.rb

module ApplicationCable
  class Channel < ActionCable::Channel::Base
    end
end
```

Okay, let's move on to writing our own Action Cable code.

First things first, setting up the instance of the Action Cable server.

## [Establishing the WebSocket Connection](#)

### [Step 1: Establish the Socket Connection: Server-Side](#)

First, we need to mount the Action Cable server on a sub-URI of our main application.

In `routes.rb`:

```
Rails.application.routes.draw do
  # Serve websocket cable requests in-process
  mount ActionCable.server => '/cable'

  resources :chatrooms, param: :slug
  resources :messages

  ...
end
```

Now, Action Cable will be listening for WebSocket requests on `ws://localhost:3000/cable`. It will do so by using the Rack socket hijacking API. When our main application is instantiated, an instance of Action Cable will also be created. Action Cable will, per our instructions in the `routes.rb` file, establish a WebSocket connection on `localhost:3000/cable`, and begin listening for socket requests on that URI.

Now that we've established the socket connection on the server-side, we need to create the client of the WebSocket connection, called the **consumer**.

### [Step 2: Establish the Socket Connection: Client-Side](#)

In `app/assets/javascripts/channels` we'll create a file: `chatrooms.js`. Here is where we will define the client-side instance of our WebSocket connection.

```
// app/assets/javascripts/channels/chatrooms.js

//= require cable
//= require_self
//= require_tree .

this.App = {};

App.cable = ActionCable.createConsumer();
```

**Note:** Make sure you require the `channels` subdirectory in your asset pipeline by adding it to your `application.js` manifest file:

```
// app/assets/javascripts/application.js

//= require_tree ./channels
```

Notice that the `ActionCable.createConsumer` function *doesn't specify the socket URI*, `ws://localhost:3000/cable`.

How does the consumer know where to connect? We'll specify the development and production socket URIs in the appropriate environment files, and pass it through to the consumer via the `action_cable_meta_tag`.

In development

```
# config/development.rb
Rails.application.configure do
```

```
config.action_cable.url = "ws://localhost:3000/cable"
end
```

The following line is included for us in the head of our application layout:

```
# app/vippews/layouts/application.html.erb

<%= action_cable_meta_tag %>
```

**Note:** The default Action Cable URI is in fact `ws://localhost:3000/cable`, so we could have gotten away without configuring the cable url in development. I wanted to expose this configuration for anyone who wants to customize it in the future.

## Building the Channel

So far, we've used Action Cable to create a persistent connection, listening for any WebSocket requests on `ws://localhost:3000/cable`. This isn't enough to get our real-time messaging feature, however. We need to define a special Messages Channel and instruct the appropriate parts of our application to broadcast to and stream from this channel.

### Step 1: Define the Channel

Defining a channel with Action Cable is easy. We'll create a file, `app/channels/messages_channel.rb`. Here we will define our channel to inherit from the `ApplicationCable::Channel` class that we described earlier on.

```
# app/channels/messages_channel.rb

class MessagesChannel < ApplicationCable::Channel
end
```

Our Messages Channel needs only one method for our purposes, the `#subscribed` method. This method is responsible for subscribing to and streaming messages that are broadcast to this channel.

```
# app/channels/messages_channel.rb
class MessagesChannel < ApplicationCable::Channel
  def subscribed
    stream_from 'messages'
  end
end
```

We'll revisit this method in a bit, and discuss how and when it is invoked. First, let's define how and when Action Cable should broadcast new messages to the channel.

### Step 2: Broadcast to the Channel

At what point in time should a new message get broadcast to the Messages Channel? Immediately after it is created and persisted to the database. So, we'll define our broadcasting code within the `#create` action of the Messages Controller.

```
# app/controllers/messages_controller.rb

class MessagesController < ApplicationController

  def create
    message = Message.new(message_params)
    message.user = current_user
    if message.save
      ActionCable.server.broadcast 'messages',
        message: message.content,
        user: message.user.username
      head :ok
    end
  end

  ...
end
```

Here we are calling the `#broadcast` method on our Action Cable server, and passing it arguments:

- `'messages'`, the name of the channel to which we are broadcasting.
- Some content that will be sent through the channel as JSON:
  - `message`, set to the content of the message we just created.
  - `user`, set to the username of the user who created the message.

**Note:** There is nothing special about the `message` or `user` key here. We can structure or label the data we want to send across the channel any way we like. As long as we tell our subscriber (more on that soon) to access that data in accordance with how we structured it here.

### Step 3: Action Cable Loves Redis

This isn't a "step", since we don't have to actually do anything (yet). But let's take a moment to look at how Action Cable pairs with Redis.



Action Cable uses Redis to send and receive messages over the channel. So, when we told our Action Cable server to `#broadcast` to `'messages'`, we were actually saying "send new messages to the `'messages'` channel maintained by Redis."

At the same time, the `#subscribed` method of our Messages Channel is *really* streaming messages sent over the `'messages'` channel maintained by Redis.

Thus, Redis acts as a data store and ensures that messages will remain in sync across instances of our application.

Action Cable will look for the Redis configuration in `Rails.root.join('config/cable.yml')`. When we generated our new Rails 5 app, we also generated a file, `config/cable.yml`, that looks like this:

```
production:
  adapter: redis
  url: redis://localhost:6379/1

development:
  adapter: async

test:
  adapter: async
```

Here, we are specifying an adapter and a URL for each environment. We'll be running Redis on localhost for now. Later, when we discuss deployment, we'll update the production environment's Redis URL.

## Defining the Channel's Subscriber

We're almost done! We just need to create a client-side subscriber to our Messages Channel.

Recall that earlier, we created our consumer with the following lines of code:

```
// app/assets/javascripts/channels/chatrooms.js

this.App = {};

App.cable = ActionCable.createConsumer();
```

Our consumer is the client-side end of our persistent WebSocket connection.

Now, we need to add a subscription to our consumer, telling it to subscribe to the Messages Channel.

Create a file, `app/assets/javascripts/channels/messages.js`. Here we will define our subscription:

```
// app/assets/javascripts/channels/messages.js

App.messages = App.cable.subscriptions.create('MessagesChannel', {
  received: function(data) {
    $("#messages").removeClass('hidden')
    return $('#messages').append(this.renderMessage(data));
  },

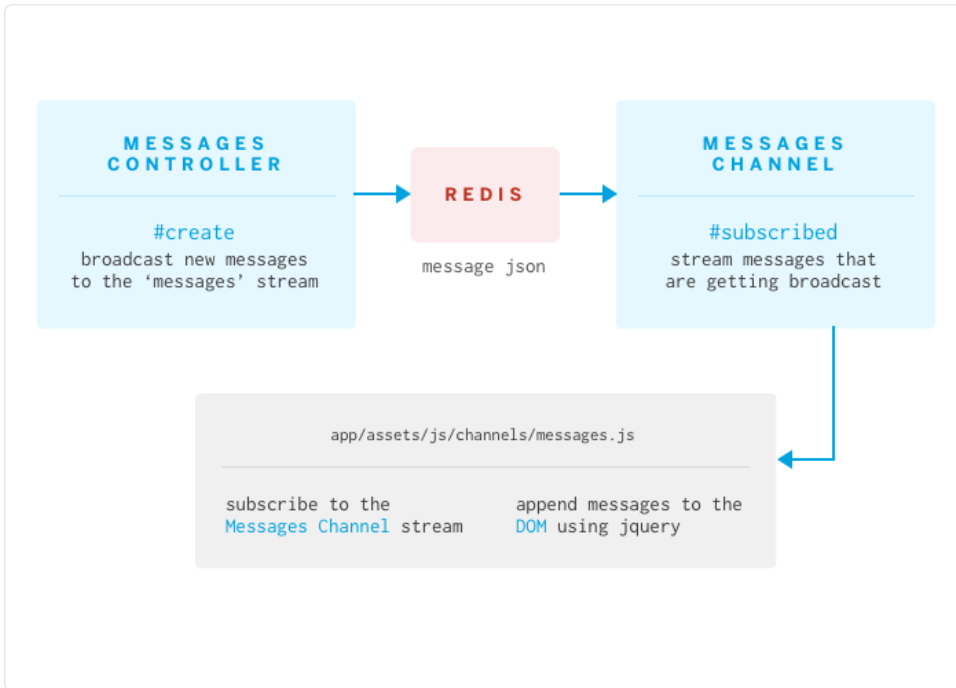
  renderMessage: function(data) {
    return "<p> <b>" + data.user + ": </b>" + data.message + "</p>";
  }
});
```

We add our new subscription to our consumer with `App.cable.subscriptions.create`. We give this function an argument of the name of the channel to which we want to subscribe, `MessagesChannel`.

When this `subscriptions.create` function is invoked, it will invoke the `MessagesChannel#subscribed` method, which is in fact a callback method.

`MessagesChannel#subscribed` streams from our messages broadcast, sending along any new messages as JSON to the client-side subscription function.

Then, the `received` function is invoked, with an argument of this new message JSON. The `received` function in turn calls a helper function that we have defined, `renderMessage`, which simply appends new messages to the DOM, using the `$("#messages")` jQuery selector, which can be found on the chatroom show page.



## Deploying our Application to Heroku

Now that our real-time messaging feature is up and running, we're ready to deploy! Deploying a Rails 5 app that uses Action Cable is simple.

We just need to configure a few things for our production environment.

### Step 1: Create Your Heroku App

First things first, though! Let's create our Heroku app:

```
heroku create action-cable-example-app
```

### Step 2: Provision the Redis To Go Addon

In the command line, run:

```
heroku addons:add redisgogo
```

Then, retrieve your Redis To Go URL and update your Action Cable Redis configuration.

```
$ heroku config --app action-cable-example | grep REDISTOGO_URL
REDISTOGO_URL: redis://redistogo:d0ed635634356d4408c1effb00bc9493@hoki.redistogo.com:9247/

# config/cable.yml

production:
  adapter: redis
  url: redis://redistogo:d0ed635634356d4408c1effb00bc9493@hoki.redistogo.com:9247/

development:
  adapter: async

test:
  adapter: async
```

### Step 3: Configure Action Cable's Production URI

We need to set the cable server's URI for production.

```
# config/environments/production.rb

config.web_socket_server_url = "wss://action-cable-example.herokuapp.com/cable"
```

**Note:** In production, we are using a *secure* WebSocket connection, wss.

### Step 4: Allowed Request Origins

Action Cable can only accept WebSocket requests from specified origins. We need to pass those origins to the Action Cable server's configuration as an array.

```
# config/environments/production.rb
```

```
config.action_cable.allowed_request_origins = ['https://action-cable-example.herokuapp.com', 'http://action-cable-example.herokuapp.com']
```

## Step 5: Deploy!

Now we're ready to `git push heroku master`. Go ahead and migrate your database and you should be good to go.

## Action Cable: Comprehensive, Sleek and Easy to Use

So how does Action Cable stack up to DHH's claims, one year later?

So far, we've seen that Action Cable runs seamlessly alongside our main Rails application. Its implementation falls right in line with the design patterns we've become so familiar with.

We mount the Action Cable server in the `routes.rb` file, alongside the rest of our routes. We write the code to broadcast new messages in the `#create` action of the Messages Controller, and we subscribe to those messages in a channel we define similarly to the manner in which we define Rails controllers.

Not only does Action Cable seamlessly integrate with the rest of our Rails application, it provides easy-to-use client-side code, making it a full stack offering.

Above all, it allows Rails developers who want real-time functionality to be totally self-reliant. We no longer need to look towards external libraries like Faye and Private Pub or implement strategies like JavaScript polling. With the addition of Action Cable, Rails is a truly integrated system with which to build a full-stack application.

Over all, Action Cable is a very welcome addition to the Rails tool kit.

---

**Sophie DeBenedetto** is a web developer and an instructor at the Flatiron School. Her first love is Ruby on Rails, although she has developed projects with and written about Rails, Ember and Phoenix. You can learn more about her recent projects by visiting her [personal site](#), checking out her [GitHub account](#), or reading her [blog](#).

[action cable](#) [sockets](#) [websockets](#) [rails 5](#) [real-time rails](#) [ruby](#)

- [Tweet](#)
- [Share](#)
- [Share](#)

---

### Related posts

- [Container-Ready Rails 5](#)
- [Where Will Ruby Go Now? Talking with Tenderlove at RailsConf](#)
- [Upgrading to Rails 5 Beta - The Hard Way](#)
- [Adding Concurrency to Rails Apps with Unicorn](#)
- [Evolution of the Heroku CLI: 2008-2017](#)

Browse the archives for [news](#) or [all blogs](#). Subscribe to the RSS feed for [news](#) or [all blogs](#).

### Products

- [Heroku Platform](#)
- [Heroku Connect](#)
- [Heroku Postgres](#)
- [Heroku Redis](#)
- [Kafka on Heroku](#)
- [Heroku Enterprise](#)
- [Heroku Teams](#)
- [Elements Marketplace](#)
- [Pricing](#)

### Resources

- [Documentation](#)
- [Blog](#)
- [Get Started](#)

### About

- [About Us](#)
- [What is Heroku](#)
- [Our Customers](#)
- [Careers](#)
- [Partners](#)
- [Elements](#)

## Support

- [Help Center](#)
- [Status](#)
- [Premium Support](#)
- [Contact Us](#)

## Subscribe to our monthly newsletter

do not fill this in  Your email address

- [RSS](#)
  - [Heroku Blog](#)
  - [Heroku News Blog](#)
  - [Heroku Engineering Blog](#)
  - [Dev Center Articles](#)
  - [Dev Center Changelog](#)
  - [Heroku Status](#)
- [Twitter](#)
  - [Heroku](#)
  - [Dev Center Articles](#)
  - [Dev Center Changelog](#)
  - [Heroku Status](#)
- [Facebook](#)
- [Instagram](#)
- [GitHub](#)
- [LinkedIn](#)

## [Heroku is a company](#)

- [heroku.com](#)
- [Terms of Service](#)
- [Privacy](#)
- [Cookies](#)
- © 2017 Salesforce.com