Using WebSockets on Heroku with Ruby

Last updated 26 August 2016

Table of Contents

This quickstart will get you going with a Sinatra application that uses a WebSocket, deployed to Heroku.

Sample code for the **demo application (https://github.com/heroku-examples/ruby-websockets-chat-demo)** is available on GitHub. Edits and enhancements are welcome. Just fork the repository, make your changes and send us a pull request.

Prerequisites

- A Heroku user account. Signup is free and instant (https://signup.heroku.com/signup/dc).
- Ruby 2.1.2+ and the Heroku CLI (https://cli.heroku.com/) (as described in the basic Ruby Getting Started guide (https://devcenter.heroku.com/articles/getting-started-with-ruby))
- Working knowledge of Rack Middleware (http://railscasts.com/episodes/151-rack-middleware)

Create WebSocket app

The sample application provides a simple example of using WebSockets with Ruby. You can clone the sample and follow along with the code as you read.

Option 1. Clone sample app

```
$ git clone git@github.com:heroku-examples/ruby-websockets-chat-demo.git
Cloning into 'ruby-websockets-chat-demo'...
remote: Counting objects: 31, done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 31 (delta 0), reused 31 (delta 0)
Receiving objects: 100% (31/31), 38.33 KiB | 0 bytes/s, done.
```

Option 2. Create a new app

```
$ mkdir ruby-websockets-chat-demo
$ cd ruby-websockets-chat-demo
```

Functionality

The demo app is a simple chat application that will open a WebSocket to the back-end. Any time a chat message is sent from the browser, it's sent to the server and then broadcasted to each connecting client and displayed on the page.

There are a few key pieces to this implementation. We're using Faye's WebSockets implementation (https://github.com/faye/faye-websocket-ruby) that provides the standard WebSockets API (http://dev.w3.org/html5/websockets/). This allows us to build a Rack middleware that responds to WebSockets.

JavaScript on the browser opens a WebSocket connection to the server and responds to a WebSocket message being received from the server to display the chat message. Let's take a look at both the back-end and front-end in more detail.

Back-end

In the sample app, we create a Rack middleware (https://github.com/heroku-examples/ruby-websockets-chat-demo/blob/33eef0f092aec7c5500f328f4adf941adc5d4836/middlewares/chat_backend.rb) to encapsulate the WebSockets logic called `ChatBackend`. For organizational purposes we're going to put all of our Rack middleware in a `middlewares` directory.

Let's walk through our `ChatBackend` middleware. We need to keep track of all the clients that are connecting to the web app, so let's setup a `@clients` array along with some basic boilerplate code:

```
# middlewares/chat_backend.rb
require 'faye/websocket'

module ChatDemo
  class ChatBackend
    KEEPALIVE_TIME = 15 # in seconds

    def initialize(app)
      @app     = app
      @clients = []
    end

    def call(env)
    end
  end
end
```

Inside the `call` method, `Faye::Websocket` allows us to detect whether the incoming request is a WebSockets request by inspecting `env`. If it is let's do our WebSockets logic. If it isn't, then we should just go through the rest of the stack (in our case this is the Sinatra app).

```
# middlewares/chat_backend.rb
def call(env)
  if Faye::WebSocket.websocket?(env)
    # WebSockets logic goes here

    # Return async Rack response
    ws.rack_response
  else
    @app.call(env)
  end
end
```

Now let's add the WebSockets code. We need to first create a new WebSockets object based off the `env`. We can do this using the `Faye::WebSocket` initializer. The options hash accepts a `ping` option which sends a ping to each active connection every X number of seconds. In our case we're going to use the `KEEPALIVE_TIME` constant we set above to 15 seconds. We want to do this because if the connection is idle for 55 seconds, Heroku will terminate the connection (https://devcenter.heroku.com/articles/http-routing#timeouts).

Initialize the WebSocket in the `call` method of your middleware with the `ping` option.

```
# middlewares/chat_backend.rb#call
ws = Faye::WebSocket.new(env, nil, {ping: KEEPALIVE_TIME })
```

The three WebSocket events that are important for this app are `open`, `message`, `close`. In each event we're going to print messages to `STDOUT` for demo purposes to see the lifecycle of WebSockets.

open

`open` gets invoked when a new connection to the server happens. For `open`, we're just going to store that a client has connected in the `@clients` array we set above.

```
# middlewares/chat_backend.rb#call
ws.on :open do |event|
  p [:open, ws.object_id]
  @clients << ws
end
```

message

`message` gets invoked when a WebSockets message is received by the server. For `message`, we need to broadcast the message to each connected client. The `event` object passed in has a `data` attribute which is the message.

```
# middlewares/chat_backend.rb#call
ws.on :message do |event|
  p [:message, event.data]
  @clients.each {|client| client.send(event.data) }
end
```

close

close gets invoked when the client closes the connection. For close , we just need to cleanup by removing the client from our store of clients.

```
# middlewarces/chat_backend.rb#call
ws.on :close do |event|
  p [:close, ws.object_id, event.code, event.reason]
  @clients.delete(ws)
  ws = nil
end
```

Front-end

The second part of this is setting up the client side to actually open the WebSockets connection with the server. First, we need to finish setting up the Sinatra app to render the pages to use.

View

In the Sinatra app, we just need to render the index view which we'll use the built in `ERB` .

```
# app.rb
require 'sinatra/base'

module ChatDemo
  class App < Sinatra::Base
    get "/" do
      erb :"index.html"
    end
  end
end
```

Sinatra stores its views in the `views` directory. Create an `index.html.erb` (https://github.com/heroku-examples/ruby-websockets-chat-demo/blob/33eef0f092aec7c5500f328f4adf941adc5d4836/views/index.html.erb) with a basic chat form.

Receive messages

Now back to WebSockets. Let's write the `public/assets/js/application.js` that gets loaded by the main page and establishes a WebSocket connection to the back-end.

```
var scheme   = "ws://";
var uri      = scheme + window.document.location.host + "/";
var ws       = new WebSocket(uri);
```

With an open WebSocket, the browser will receive messages. These messages are structured as a JSON response with two keys: `handle` (user's handle) and `text` (user's message). When the message is received, it's inserted as a new entry in the page.

```
// public/assets/js/application.js
ws.onmessage = function(message) {
  var data = JSON.parse(message.data);
  $("#chat-text").append("<div class='panel panel-default'><div class='panel-heading'>" + data.
  $("#chat-text").stop().animate({
    scrollTop: $('#chat-text')[0].scrollHeight
  }, 800);
};
```

Send messages

The now that the page can receive messages from the server, we need a way to actually send messages. We'll override the form submit button to grab the the values from the form and send them as a JSON message over the WebSocket to the server.

```
// public/assets/js/application.js
$("#input-form").on("submit", function(event) {
  event.preventDefault();
  var handle = $("#input-handle")[0].value;
  var text   = $("#input-text")[0].value;
  ws.send(JSON.stringify({ handle: handle, text: text }));
  $("#input-text")[0].value = "";
});
```

Your final `application.js` file should now define the complete send and receive functionality (https://github.com/heroku-examples/ruby-websockets-chat-demo/blob/33eef0f092aec7c5500f328f4adf941adc5d4836/public/assets/js/application.js).

Local execution

To wire up the application, we'll use `config.ru` which sets up `Rack::Builder`. This tells Rack and the server how to load the application. We want to setup our WebSockets middleware, so requests go through this first.

```
# config.ru
require './app'
require './middlewares/chat_backend'

use ChatDemo::ChatBackend

run ChatDemo::App
```

We need to download and fetch the dependencies, so we need to setup a `Gemfile`.

```
# Gemfile
source "https://rubygems.org"

ruby "2.0.0"

gem "faye-websocket"
gem "sinatra"
gem "puma"
```

And install all dependencies.

```
$ bundle install
```

We'll now setup a Procfile (https://devcenter.heroku.com/articles/procfile), so we have a documented way of running our web service. We'll be using the puma webserver (http://puma.io/) which is one of the webservers (https://github.com/faye/faye-websocket-ruby/blob/master/README.md) supported by `faye-websocket` .

```
web: bundle exec puma -p $PORT
```

Run the app locally.

```
$ heroku local web
Puma starting in single mode...
* Version 2.6.0, codename: Pantsuit Party
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://0.0.0.0:5000
Use Ctrl-C to stop
```

Open the app at localhost:5000 (http://localhost:5000) and enter a chat message. You should see the following in the server output:

```
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET / HTTP/1.1" 200 1430 0.0115
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET /assets/css/application.css HTTP/1.1" 304 - 0.0164
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET /assets/css/bootstrap.min.css HTTP/1.1" 304 - 0.0046
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET /assets/js/jquery-2.0.3.min.js HTTP/1.1" 304 - 0.0007
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET /assets/js/application.js HTTP/1.1" 200 716 0.0063
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET / HTTP/1.1" HIJACKED -1 0.0059
[:open, 7612540]
127.0.0.1 - - [03/Oct/2013 17:16:25] "GET /favicon.ico HTTP/1.1" 404 490 0.0012
[:message, "{\"handle\":\"hone\",\"text\":\"test\"}"]
```

Deploy

After the app is running locally, it's time to deploy it to Heroku. If you haven't done so already put your application into a git repository:

```
$ git init
$ git add .
$ git commit -m "Ready to deploy"
```

Create the app to deploy to on Heroku.

```
$ heroku create
Creating limitless-ocean-5045... done, stack is cedar-14
http://limitless-ocean-5045.herokuapp.com/ | git@heroku.com:limitless-ocean-5045.git
Git remote heroku added
```

Deploy your app to Heroku with `git push`:

```
$ git push heroku master
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 557 bytes, done.
Total 5 (delta 3), reused 0 (delta 0)

-----> Ruby/Rack app detected
-----> Using Ruby version: ruby-2.1.2
-----> Installing dependencies using Bundler version 1.6.3
       Running: bundle install --without development:test --path vendor/bundle --binstubs vendor/bundle
       Installing eventmachine (1.0.3)
       Installing websocket-driver (0.3.0)
       Installing faye-websocket (0.7.0)
       Installing rack (1.5.2)
       Installing puma (2.6.0)
       Installing rack-protection (1.5.0)
       Installing redis (3.0.4)
       Installing tilt (1.4.1)
       Installing sinatra (1.4.3)
       Installing bundler (1.3.2)
       Your bundle is complete! It was installed into ./vendor/bundle
       Cleaning up the bundler cache.
-----> Discovering process types
       Procfile declares types     -> web
       Default types for Ruby/Rack -> console, rake

-----> Compiled slug size: 26.8MB
-----> Launching... done, v3
       http://limitless-ocean-5045.herokuapp.com deployed to Heroku
```

Congratulations! Your web app should now be up and running on Heroku. use `heroku open` to open it in your browser.

Scaling

Now that you have deployed your app and understand the very basics we'll want to expand the app to be a bit more robust. This won't encompass everything you'd want to do before going public, but it will get you along the a path where you'll be thinking about the right things.

Currently, if you scale your app to more than a single dyno, not everyone will get all messages. Due to the stateless nature of the current app, clients connected to dyno #1 won't get messages sent by clients connected to dyno #2. We can solve this in the chat example by storing the message state in a global storage system like Redis. This allows messages to be sent to every dyno connected to Redis and is explained in more detail in our WebSockets Application Architecture Section (https://devcenter.heroku.com/articles/websockets#application-architecture).

Add a Redis add-on (https://elements.heroku.com/addons/):

```
$ heroku addons:create rediscloud:20
Adding rediscloud:20 on limitless-ocean-5045... v4 (free)
```

Make sure you have Redis setup locally (https://devcenter.heroku.com/articles/queuing-ruby-resque#install-locally). We'll be using the `redis` gem to interface with Redis from the ruby app. Like above, we'll need to edit the `Gemfile` to configure app dependencies. Add this line to the bottom of your `Gemfile` .

```
gem 'redis'
```

Install the dependencies.

```
$ bundle install
```

We need to setup each dyno to use Redis' pubsub system (http://robots.thoughtbot.com/post/6325247416/redis-pub-sub-how-does-it-work). All the dynos will subscribe to the same channel `chat-demo` and wait for messages. When each server gets a message, it can publish it to the clients connected.

```
require 'redis'

module ChatDemo
  class ChatBackend
    ...
    CHANNEL        = "chat-demo"

    def initialize(app)
      ...
      uri      = URI.parse(ENV["REDISCLOUD_URL"])
      @redis   = Redis.new(host: uri.host, port: uri.port, password: uri.password)
      Thread.new do
        redis_sub = Redis.new(host: uri.host, port: uri.port, password: uri.password)
        redis_sub.subscribe(CHANNEL) do |on|
          on.message do |channel, msg|
            @clients.each {|ws| ws.send(msg) }
          end
        end
      end
    end
    ...
end
```

We need to do the subscribe in a separate thread, because subscribe is blocking function, so it will stop the executon flow to wait for a message. In addition, a second redis connection is needed since once a subscribe command has been made on a connection, the connection can only unsubscribe or receive messages.

Now when the server receives a message, we want to publish it with Redis to the channel instead of sending it to the browser. This way every dyno will get notified of it, so every client can receive the message. Change `middlewares/chat_backend#call`:

```
ws.on :message do |event|
  p [:message, event.data]
  @redis.publish(CHANNEL, event.data)
end
```

All these steps can be viewed in this commit (https://github.com/heroku-examples/ruby-websockets-chat-demo/commit/0f5fc0e087c31a400b5f739a2bca0f3163403c7f).

Security

Currently, the application is exposed and vulnerable to many attacks. Please refer to WebSockets Security (https://devcenter.heroku.com/articles/websocket-security) for more general guidelines on securing your WebSockets application. The example app on github (https://github.com/heroku-examples/ruby-websockets-chat-demo) sets up WSS (https://github.com/heroku-examples/ruby-websockets-chat-demo/commit/9b4d0b6866c86e88dd5edc9daa9cbee389a995e8) and sanitizes input (https://github.com/heroku-examples/ruby-websockets-chat-demo/commit/ed4b973744f1b277765d36e2c95a709e116f3d03).

Using with Rails

In this sample app we created Rack middleware alongside a Sinatra app. Using the same middleware in a Rails app is trivial.

Copy the existing `ChatBackend` middleware to `app/middleware/chat_backend.rb` in your Rails project. Then insert the middleware into your stack, defined in `config/application.rb` :

```
require 'chat_backend'
config.middleware.use ChatDemo::ChatBackend
```

WebSocket requests will now be handled by the custom chat middleware, embedded within your Rails app.