
faye-websocket

This is a general-purpose WebSocket implementation extracted from the Faye project. It provides classes for easily building WebSocket servers and clients in Ruby. It does not provide a server itself, but rather makes it easy to handle WebSocket connections within an existing Rack application. It does not provide any abstraction other than the standard WebSocket API.

It also provides an abstraction for handling EventSource connections, which are one-way connections that allow the server to push data to the client. They are based on streaming HTTP responses and can be easier to access via proxies than WebSockets.

The following web servers are supported. Other servers that implement the `rack.hijack` API should also work.

- Goliath
- Phusion Passenger ≥ 4.0 with nginx ≥ 1.4
- Puma ≥ 2.0
- Rainbows
- Thin

Installation

```
1 $ gem install faye-websocket
```

Handling WebSocket connections in Rack

You can handle WebSockets on the server side by listening for requests using the `Faye::WebSocket.websocket?` method, and creating a new socket for the request. This socket object exposes the usual WebSocket methods for receiving and sending messages. For example this is how you'd implement an echo server:

```
1 # app.rb
2 require 'faye/websocket'
3
4 App = lambda do |env|
5   if Faye::WebSocket.websocket?(env)
6     ws = Faye::WebSocket.new(env)
7
8     ws.on :message do |event|
9       ws.send(event.data)
10    end
11  end
```

```

12     ws.on :close do |event|
13       p [:close, event.code, event.reason]
14       ws = nil
15     end
16
17     # Return async Rack response
18     ws.rack_response
19
20   else
21     # Normal HTTP request
22     [200, { 'Content-Type' => 'text/plain' }, ['Hello']]
23   end
24 end

```

Note that under certain circumstances (notably a draft-76 client connecting through an HTTP proxy), the WebSocket handshake will not be complete after you call `Faye::WebSocket.new` because the server will not have received the entire handshake from the client yet. In this case, calls to `ws.send` will buffer the message in memory until the handshake is complete, at which point any buffered messages will be sent to the client.

If you need to detect when the WebSocket handshake is complete, you can use the `onopen` event.

If the connection's protocol version supports it, you can call `ws.ping()` to send a ping message and wait for the client's response. This method takes a message string, and an optional callback that fires when a matching pong message is received. It returns `true` if and only if a ping message was sent. If the client does not support ping/pong, this method sends no data and returns `false`.

```

1 ws.ping 'Mic check, one, two' do
2   # fires when pong is received
3 end

```

Using the WebSocket client

The client supports both the plain-text `ws` protocol and the encrypted `wss` protocol, and has exactly the same interface as a socket you would use in a web browser. On the wire it identifies itself as `hybi-13`.

```

1 require 'faye/websocket'
2 require 'eventmachine'
3
4 EM.run {
5   ws = Faye::WebSocket::Client.new('ws://www.example.com/')
6
7   ws.on :open do |event|
8     p [:open]
9     ws.send('Hello, world!')

```

```
10 end
11
12 ws.on :message do |event|
13   p [:message, event.data]
14 end
15
16 ws.on :close do |event|
17   p [:close, event.code, event.reason]
18   ws = nil
19 end
20 }
```

The WebSocket client also lets you inspect the status and headers of the handshake response via its `status` and `headers` methods.

To connect via a proxy, set the `proxy` option to the HTTP origin of the proxy, including any authorization information and custom headers you require:

```
1 ws = Faye::WebSocket::Client.new('ws://www.example.com/', [], {
2   :proxy => {
3     :origin => 'http://username:password@proxy.example.com',
4     :headers => { 'User-Agent' => 'ruby' }
5   }
6 })
```

Subprotocol negotiation

The WebSocket protocol allows peers to select and identify the application protocol to use over the connection. On the client side, you can set which protocols the client accepts by passing a list of protocol names when you construct the socket:

```
1 ws = Faye::WebSocket::Client.new('ws://www.example.com/', ['irc', 'amqp'])
```

On the server side, you can likewise pass in the list of protocols the server supports after the other constructor arguments:

```
1 ws = Faye::WebSocket.new(env, ['irc', 'amqp'])
```

If the client and server agree on a protocol, both the client- and server-side socket objects expose the selected protocol through the `ws.protocol` property.

Protocol extensions

faye-websocket is based on the websocket-extensions framework that allows extensions to be negotiated via the `Sec-WebSocket-Extensions` header. To add extensions to a connection, pass an array of extensions to the `:extensions` option. For example, to add permessage-deflate:

```
1 require 'permessage_deflate'
2
3 ws = Faye::WebSocket.new(env, [], :extensions => [PermessageDeflate])
```

Initialization options

Both the server- and client-side classes allow an options hash to be passed in at initialization time, for example:

```
1 ws = Faye::WebSocket.new(env, protocols, options)
2 ws = Faye::WebSocket::Client.new(url, protocols, options)
```

`protocols` as an array of subprotocols as described above, or `nil`. `options` is an optional hash containing any of these keys:

- `:extensions` - an array of websocket-extensions compatible extensions, as described above
- `:headers` - a hash containing key-value pairs representing HTTP headers to be sent during the handshake process
- `:max_length` - the maximum allowed size of incoming message frames, in bytes. The default value is $2^{26} - 1$, or 1 byte short of 64 MiB.
- `:ping` - an integer that sets how often the WebSocket should send ping frames, measured in seconds
- `:tls` - a hash containing key-value pairs for specifying TLS parameters. These are passed along to EventMachine and you can find more details [here](#)

Secure sockets

Starting with version 0.11.0, `Faye::WebSocket::Client` will verify the server certificate for `wss` connections. This is not the default behaviour for EventMachine's TLS interface, and so our defaults for the `:tls` option are a little different.

First, `:verify_peer` is enabled by default. Our implementation checks that the chain of certificates sent by the server is trusted by your root certificates, and that the final certificate's hostname matches the hostname in the request URL.

By default, we use your system's root certificate store by invoking `OpenSSL::X509::Store#set_default_paths`. If you want to use a different set of root certificates, you can pass them via the `:root_cert_file` option, which takes a path or an array of paths to the certificates you want to use.

```
1 ws = Faye::WebSocket::Client.new('wss://example.com/', [], :tls => {
2   :root_cert_file => ['path/to/certificate.pem']
3 })
```

If you want to switch off certificate verification altogether, then set `:verify_peer` to **false**.

```
1 ws = Faye::WebSocket::Client.new('wss://example.com/', [], :tls => {
2   :verify_peer => false
3 })
```

WebSocket API

Both the server- and client-side `WebSocket` objects support the following API:

- **on(:open){ |event| }** fires when the socket connection is established. Event has no attributes.
- **on(:message){ |event| }** fires when the socket receives a message. Event has one attribute, **data**, which is either a `String` (for text frames) or an `Array` of unsigned integers, i.e. integers in the range 0 . . 255 (for binary frames).
- **on(:error){ |event| }** fires when there is a protocol error due to bad data sent by the other peer. This event is purely informational, you do not need to implement error recovery.
- **on(:close){ |event| }** fires when either the client or the server closes the connection. Event has two optional attributes, **code** and **reason**, that expose the status code and message sent by the peer that closed the connection.
- **send(message)** accepts either a `String` or an `Array` of byte-sized integers and sends a text or binary message over the connection to the other peer; binary data must be encoded as an `Array`.
- **ping(message, &callback)** sends a ping frame with an optional message and fires the callback when a matching pong is received.
- **close(code, reason)** closes the connection, sending the given status code and reason text, both of which are optional.
- **version** is a string containing the version of the `WebSocket` protocol the connection is using.
- **protocol** is a string (which may be empty) identifying the subprotocol the socket is using.

Handling EventSource connections in Rack

EventSource connections provide a very similar interface, although because they only allow the server to send data to the client, there is no `onmessage` API. EventSource allows the server to push text messages to the client, where each message has an optional event-type and ID.

```
1 # app.rb
2 require 'faye/websocket'
3
4 App = lambda do |env|
5   if Faye::EventSource.eventsource?(env)
6     es = Faye::EventSource.new(env)
7     p [:open, es.url, es.last_event_id]
8
9     # Periodically send messages
10    loop = EM.add_periodic_timer(1) { es.send('Hello') }
11
12    es.on :close do |event|
13      EM.cancel_timer(loop)
14      es = nil
15    end
16
17    # Return async Rack response
18    es.rack_response
19
20  else
21    # Normal HTTP request
22    [200, { 'Content-Type' => 'text/plain' }, ['Hello']]
23  end
24 end
```

The `send` method takes two optional parameters, `:event` and `:id`. The default event-type is `'message'` with no ID. For example, to send a `notification` event with ID 99:

```
1 es.send('Breaking News!', :event => 'notification', :id => '99')
```

The `EventSource` object exposes the following properties:

- `url` is a string containing the URL the client used to create the EventSource.
- `last_event_id` is a string containing the last event ID received by the client. You can use this when the client reconnects after a dropped connection to determine which messages need resending.

When you initialize an EventSource with `Faye::EventSource.new`, you can pass configuration options after the `env` parameter. Available options are:

- `:headers` is a hash containing custom headers to be set on the EventSource response.

-
- **:retry** is a number that tells the client how long (in seconds) it should wait after a dropped connection before attempting to reconnect.
 - **:ping** is a number that tells the server how often (in seconds) to send 'ping' packets to the client to keep the connection open, to defeat timeouts set by proxies. The client will ignore these messages.

For example, this creates a connection that allows access from any origin, pings every 15 seconds and is retryable every 10 seconds if the connection is broken:

```
1 es = Faye::EventSource.new(es,  
2   :headers => { 'Access-Control-Allow-Origin' => '*' },  
3   :ping    => 15,  
4   :retry   => 10  
5 )
```

You can send a ping message at any time by calling `es.ping`. Unlike WebSocket the client does not send a response to this; it is merely to send some data over the wire to keep the connection alive.

Running your socket application

The following describes how to run a WebSocket application using all our supported web servers.

Running the app with Thin

If you use Thin to serve your application you need to include this line after loading `faye/websocket`:

```
1 Faye::WebSocket.load_adapter('thin')
```

Thin can be started via the command line if you've set up a `config.ru` file for your application:

```
1 $ thin start -R config.ru -p 9292
```

Or, you can use `rackup`. In development mode, this adds middlewares that don't work with async apps, so you must start it in production mode:

```
1 $ rackup config.ru -s thin -E production -p 9292
```

It can also be started using the `Rack::Handler` interface common to many Ruby servers. You can configure Thin further in a block passed to `run`:

```
1 require 'eventmachine'  
2 require 'rack'  
3 require 'thin'
```

```
4 require './app'
5
6 Faye::WebSocket.load_adapter('thin')
7
8 thin = Rack::Handler.get('thin')
9
10 thin.run(App, :Port => 9292) do |server|
11   # You can set options on the server here, for example to set up SSL:
12   server.ssl_options = {
13     :private_key_file => 'path/to/ssl.key',
14     :cert_chain_file  => 'path/to/ssl.crt'
15   }
16   server.ssl = true
17 end
```

Running the app with Passenger

faye-websocket requires either Passenger for Nginx or Passenger Standalone. Apache doesn't work well with WebSockets at this time. You do not need any special configuration to make faye-websocket work, it should work out of the box on Passenger provided you use at least Passenger 4.0.

However, you do need to insert the following code in `config.ru` for optimal WebSocket performance in Passenger. This is documented in the Passenger manual.

```
1 if defined?(PhusionPassenger)
2   PhusionPassenger.advertised_concurrency_level = 0
3 end
```

Run your app on Passenger for Nginx by creating a virtual host entry which points to your app's "public" directory:

```
1 server {
2   listen 9292;
3   server_name yourdomain.local;
4   root /path-to-your-app/public;
5   passenger_enabled on;
6 }
```

Or run your app on Passenger Standalone:

```
1 $ passenger start -p 9292
```

More information can be found on the Passenger website.

Running the app with Puma

Puma has a command line interface for starting your application:

```
1 $ puma config.ru -p 9292
```

Or, you can use `rackup`. In development mode, this adds middlewares that don't work with async apps, so you must start it in production mode:

```
1 $ rackup config.ru -s puma -E production -p 9292
```

Running the app with Rainbows

If you're using version 4.4 or lower of Rainbows, you need to run it with the EventMachine backend and enable the adapter. Put this in your `rainbows.conf` file:

```
1 Rainbows! { use :EventMachine }
```

And make sure you load the adapter in your application:

```
1 Faye::WebSocket.load_adapter('rainbows')
```

Version 4.5 of Rainbows does not need this adapter.

You can run your `config.ru` file from the command line. Again, `Rack::Lint` will complain unless you put the application in production mode.

```
1 $ rainbows config.ru -c path/to/rainbows.conf -E production -p 9292
```

Running the app with Goliath

If you use Goliath to server your application you need to include this line after loading `faye/websocket`:

```
1 Faye::WebSocket.load_adapter('goliath')
```

Goliath can be made to run arbitrary Rack apps by delegating to them from a `Goliath::API` instance. A simple server looks like this:

```
1 require 'goliath'
2 require './app'
3 Faye::WebSocket.load_adapter('goliath')
4
5 class EchoServer < Goliath::API
6   def response(env)
```

```
7     App.call(env)
8   end
9 end
```

Faye::WebSocket can also be used inline within a Goliath app:

```
1 require 'goliath'
2 require 'faye/websocket'
3 Faye::WebSocket.load_adapter('goliath')
4
5 class EchoServer < Goliath::API
6   def response(env)
7     ws = Faye::WebSocket.new(env)
8
9     ws.on :message do |event|
10       ws.send(event.data)
11     end
12
13     ws.rack_response
14   end
15 end
```