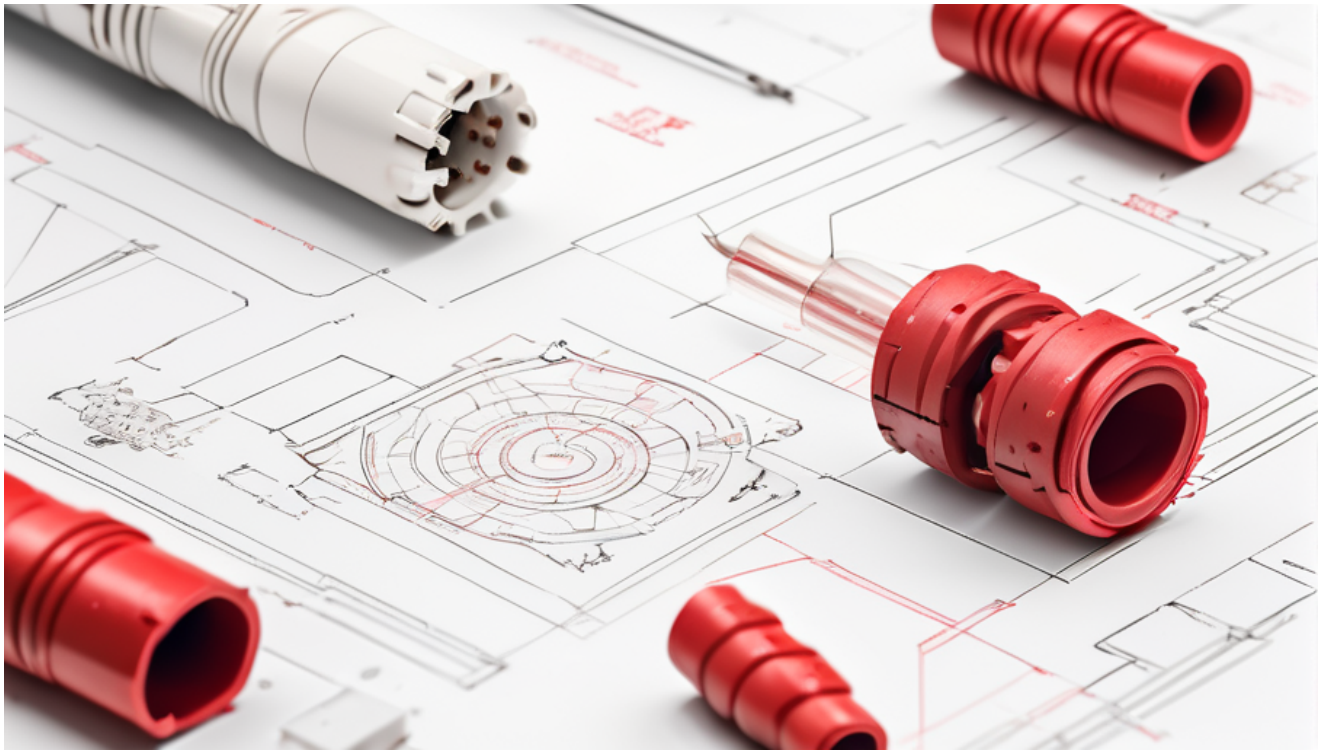


I knew what Action Cable was about and roughly how it worked since it was released. But lately I've developed a much deeper understanding of it because I had to extend it and explain its internals to others.

The following is how I explained Action Cable to myself. I took a top-down approach, starting at what I could observe Action Cable doing in my browser and following the code until I found the implementation, which I then deconstructed to understand why it does what it does.

This was a along journey down a deep rabbit hole, but it was worth it. At the end I now understand how and why Action Cable does what it does, and appreciate how simple and elegant it actually is.

From HTTP to WebSocket

Action Cable is a framework that allows your server to push messages to the browser, or any other client.

Usually your server can only respond to requests that a client, like your browser, makes to it. But Action Cable opens a WebSocket between the client and the server through which the server can push messages to the client at any time.

The client can also push messages to the server through that WebSocket. But that's in my opinion less interesting as we could do that without Action Cable.

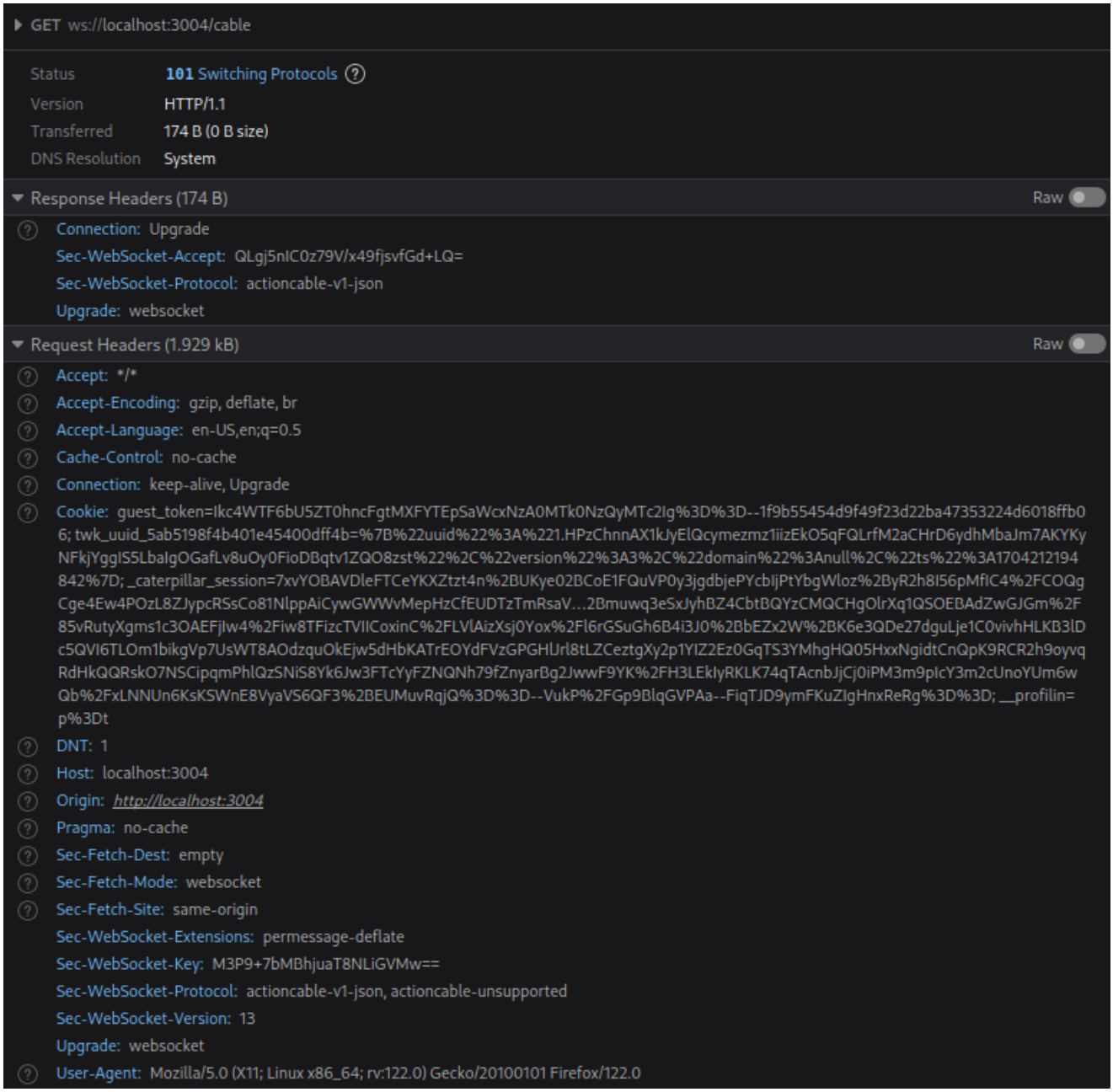
How does the client connect to Action Cable's WebSocket?

When anyone or anything wants to connect to your Rails app via Action Cable they first have to make a HTTP request to "/cable" and after a bit of back and forth out pops a WebSocket.

But how does a HTTP request become a Websocket?

Back in 1997 HTTP version 1.1 was released. It's very similar to HTTP 1.0 with a few additions, the one that's important for WebSockets is the addition of status 101 aka. Switching protocols and the *"Upgrade"* header. The intent of status 101 was to enable browsers to gracefully transition from HTTP 1.x to an eventual HTTP 2 or some other protocol that would be incompatible with HTTP 1.

And that's exactly what a browser does when it connects to a WebSocket. It first makes a request with the *"Upgrade"* header set to "websocket" which tells the server "hey, I want to switch to using WebSockets for this request". If the server supports WebSockets it responds with status 101 indicating that the upgrade was accepted and that, from now on, it will communicate using the WebSocket protocol on that request's connection.



But why do we need the HTTP request in the first place? Can't we just connect directly via the WebSocket protocol since we already know that the server supports it?

The genius of WebSockets lies in the initial HTTP request. Because of that initial HTTP request we have access to all the browser's cookies that are sent over, so we can use them to figure out who wants to open a WebSocket, if they are allowed to do that, and we can remember who the socket is for.

So Action Cable converts HTTP requests to WebSockets?

Yes! It's a WebSocket server, but it also does so much more.

Connections

So someone's connected to our WebSocket, but how do we know who we are talking to and how can we send messages to them?

In Action Cable, each open WebSocket has a corresponding Connection object. The connection object is responsible for keeping track of who the WebSocket is for and figuring out if the client is allowed to open a WebSocket.

When a client makes the initial HTTP request to *"/cable"*, Action Cable will take the request's headers, cookies, URL and create a Connection object from it.

If it exists, the Connection object will be an instance of *"ApplicationCable::Connection"*. If that class doesn't exist it will be an instance of *"ActionCable::Connection::Base"*. You can change the class of the connection by setting *"config.action_cable.connection_class = ->{ WhateverClassIWant }"*.

When a connection is established the *"#connect"* method on our *"ApplicationCable::Connection"* object will get called. In that method you'd usually want to figure out who opened the connection, and reject it if they aren't allowed to open it.

```
#!/usr/bin/ruby
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    attr_accessor :current_person

    def connect
          Rails.logger.debug "Someone wants to connect via WebSocket!"
          set_current_person

          unless current_person
            Rails.logger.debug "I don't know who this is. Closing the
WebSocket."
            reject_unauthorized_connection
          end
```

```
      Rails.logger.debug "It's, #{current_person.name}"
    end

    private

    def set_current_person
      person =
        # We can access the param of the HTTP request
        Person.find_by(token: request.params[:token]) ||
        # We can access the session of the HTTP request
        Person.find_by(token: request.session[:person_id])

      unless person
        # We can access the cookies of the HTTP request
        session = Session.find_by(id: cookies.encrypted[:session_id])
        person = session&.person
      end

      self.current_person = person
    end
  end
end
```

There is also a *"#disconnect"* method that gets called when a WebSocket is closed. It's useful if you want to set something up in *"#connect"* and then tear it down when the WebSocket closes. E.g. if you'd want to count the number of open connections that a person has, you could do something like this.

```
#!/usr/bin/ruby
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    attr_accessor :current_person

    def connect
      set_current_person
      reject_unauthorized_connection unless current_person

      current_person.increment!(:connection_count)
    end

    def disconnect
      current_person&.decrement!(:connection_count)
    end

    private

    # ...
```

```
  end
end
```

Now you know who the WebSocket connection is for, but Rails doesn't. To tell Rails which attr_accessor(s) we'll use for identification we have to change our *"attr_accessor"* to *"identified_by"*, which is a class method that Action Cable provides.

```
#!/usr/bin/ruby
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    #attr_accessor :current_person # CHANGE THIS
    identified_by :current_person # TO THIS

    def connect
          set_current_person
          reject_unauthorized_connection unless current_person
    end

    private

    # ...
  end
end
```

Ok, now we've got a connection and we know who it's for, how do we send a message? Do we have to send some kind of JSON or XML or whatnot?

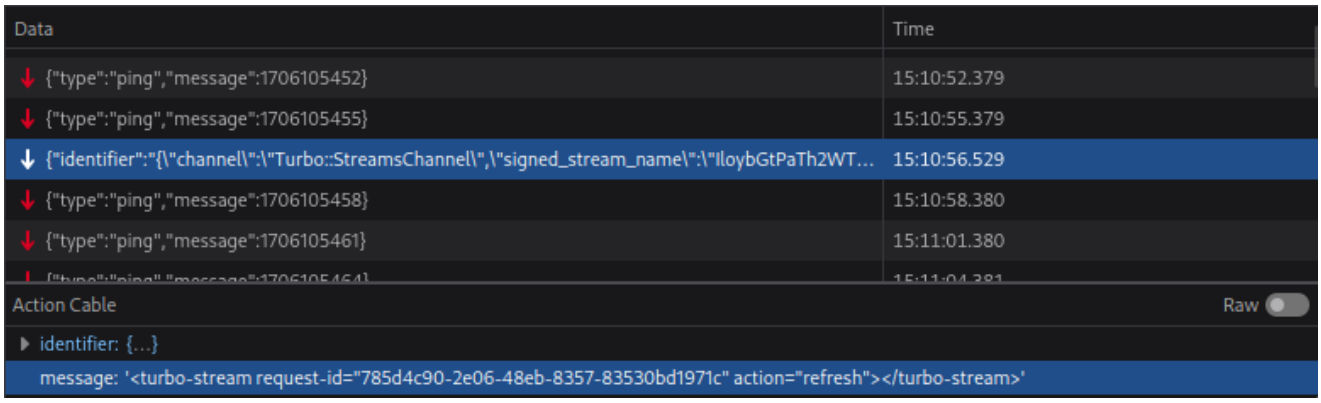
The Wild WebSocket West

We are used to HTTP in our Rails apps. With it we can transfer text, images, audio, video and more from our server to the browser. But WebSockets are the wild west of the Web.

As the "socket" part implies, a WebSocket is similar to a Unix, TCP or UDP socket in that you can transfer whatever you like over it and in any fashion you'd like. It's up to you to implement a protocol between your server and the browser on top of your WebSocket.

Luckily we don't have to implement anything as **Action Cable is not only a WebSocket server but also a protocol.**

When you send a message to a client - or vice versa - the the object you are sending will first be turned into a JSON-compatible object like a String, a Number, a Hash, or an Array. Then it will be wrapped inside a Hash under a key called "message" together with some additional things I'll explain in a bit. Then that Hash will be dumped as JSON and sent as an UTF-8 encoded String, byte for byte through the WebSocket. On the other end the message is assembled into a String again and parsed into an object.



Why do we need the wrapper Hash?

The wrapper Hash allows us to have different types of messages. The protocol has three types of messages that a client can send to the server:

- subscribe
- unsubscribe
- message

And six kinds of messages that the server can send to the client:

- welcome
- disconnect
- ping
- confirm_subscription
- reject_subscription
- message

The "welcome" message is the simplest of them all. It's just a Hash with a key "type" that holds the value "welcome". All messages sent from the server to a client have a "type" field. This is always the first message that a server sends to any client that connects to it. It's used to indicate to the client that the WebSocket is functional.

```
{
  "type": "welcome"
}
```

The "ping" message is used to keep a heartbeat. It's sent to all connected clients every 3 seconds so that the connection doesn't become stale and therefore terminated by some load balance in the middle. And it enables clients to detect when they have disconnected from the server. If the client doesn't receive a heartbeat message within 6 seconds - that's 2 heartbeats - it will try to reconnect. In addition to the "type" field, a ping message will also have a "message" field which holds the Unix timestamp of the moment the message was sent. This timestamp isn't used in the official client implementation.

```
{
  "type": "ping",
  "message": 1705848059
}
```

The "disconnect" message instructs a client to disconnect from the server. In addition to the "type" field it has a "reconnect" field which holds a Boolean. If the value of this field is true, the client should immediately try to reconnect after it disconnects. This is useful in case an error occurs and the connection has to be restarted. And in addition to the "reconnect" field there is a "reason" field which holds a String that gives you a brief explanation why your client is being disconnected.

There are four possible disconnect reasons:

- *unauthorized* - sent when "reject_unauthorized_connection" is called
- *invalid_request* - sent when the initial HTTP request was malformed
- *server_restart* - sent when the server is about to restart
- *remote* - send when the client is kicked for whatever reason

```
{
  "type": "disconnect",
  "reconnect": true,
  "reason": "server_restart"
}
```

To explain the *"message"*, *"subscribe"*, *"confirm_subscription"*, *"reject_subscription"* & *"unsubscribe"* messages I first have to explain Channels.

Channels

What are Channels?

Channels are Action Cable's controllers.

In a regular HTTP request you'd specify which method you want to call on which Controller by setting the request's path and method. E.g. if you'd want to call the *"create"* method of the *"ArticlesController"* you'd make a POST request to "/articles" and pass any parameters you'd like the new Article object to have.

```
#!/usr/bin/ruby
class ArticlesController
  # POST /articles
  def create
    if Article.create(params.require(:article).permit(:title, :content))
      redirect_to action: :show, status: :see_other
```

```
    else
      render :new, status: :unprocesssable_entity
    end
  end
end
```

In Action Cable you specify which method you want to call on which Channel. To do that, you first have to subscribe to to a Channel and then you have to send a message to it.

```
#!/usr/bin/ruby
class ChatChannel < ApplicationCable::Channel
  def post_message(data)
          Chat::Message.create(
            content: data[:content],
            poster: current_person
          )
  end
end
```

(Note that we get access to *"current_person"* because we used *"identified_by"* in our Connection object)

Now if you want to invoke *"ChatChannel#post"* you first have to subscribe to the *"ChatChannel"* with a *"subscribe"* message.

The "subscribe" message has a *"command"* field instead of a *"type"* field. All messages sent from the client to the server have a *"command"* field. In addition to that it also has an *"identifier"* field which holds a Hash. The identifier Hash has at least a *"channel"* field which is the class name of the channel. But it can also have other, user defined, values which will become available in the Channel object through the *"params"* method.

```
{
  "command": "subscribe",
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  }
}
```

These params can be used however you'd like. E.g.

```
#!/usr/bin/ruby
class ChatChannel < ApplicationCable::Channel
  def post_message(data)
          current_person
            .chat_rooms
```

```
            .find_by(name: params[:room_name]) # PARAMS FROM THE SUBSCIBE
MESSAGE
            &.messages
            &.create(content: data[:content])
  end
end
```

When someone subscribes to a Channel the *"subscribe"* method on the channel will be called. In it you can do various things, one of which is to authorize the subscription. Let's say that you want to reject subscriptions for chat rooms that a person isn't a member of, you could do something like this

```
#!/usr/bin/ruby
class ChatChannel < ApplicationCable::Channel
  def subscribe
    @chat_room = current_person
          .chat_rooms
          .find_by(name: params[:room_name])

        reject unless @chat_room
  end

  def post_message(data)
        @chat_room.messages.create(content: data[:content])
  end
end
```

If the *"subscribe"* method doesn't call *"reject"* in some cases the server will respond with a *"confirm_subscription"* message.

The *"confirm_subscription"* message has a *"type"*, and an *"identifier"* field. The identifier holds the same value that was sent in the original *"subscribe"* message.

```
{
  "type": "confirm_subscription",
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  }
}
```

If *"reject"* was called then the server will respond with a *"reject_subscription"* message.

The "reject_subscription" message has the same format as the "confirm_subscription" message.

```
{
  "type": "reject_subscription",
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  }
}
```

Now that the client is subscribed they can invoke an action on the Channel using the regular message type.

The "message" type messages has two variants - one for messages sent from the server to the client, and another for messages sent from the client to the server.

If the client is sending the message, then the message will contain a *"command"* field with the value "message", an *"identifier"* and a *"data"* field. Again, the identifier holds the same value that was sent in the original *"subscribe"* message. While data can be anything depending on what the client has sent.

```
{
  "command": "message",
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  },
  "data": "Hello, Zagreb!"
}
```

When the channel receives a message, it will check if it implements a *"receive"* method and if it does it will invoke it and pass the message's data to it.

Though, if the client sent over a Hash, then the Channel will check if the data Hash contains the key "action" with a String value. If it does, and the Channel implements a method with the same name, then it will be invoked to process the message's data.

```
{
  "command": "message",
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  },
  "data": {
    "action": "post_message",
    "content": "Hello, Zagreb!"
  }
}
```

I'll explain the server-to-client message format in a moment.

Now we know how to send messages from the client to Action Cable. But how do we send messages from Action Cable to the client?

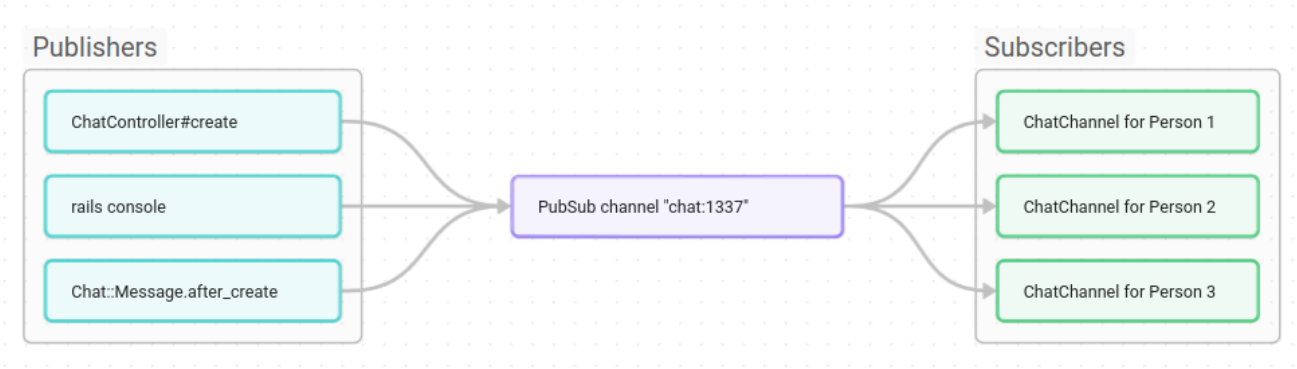
This is in my opinion the more interesting part. To send something we first have to create a stream to which we can publish messages to.

A stream is a PubSub channel. I *guess* it's called a stream in Action Cable because, semantically, this PubSub channel acts like a stream of messages for the client.

What is a PubSub channel?

PubSub (short for Publish-subscribe) is a common programming pattern for sending messages between objects. An object can subscribe to messages from, or publish messages to, a channel (sometimes called a topic). When one object publishes a message to a channel, all objects subscribed to that channel receive it. The message can be anything - a String, a Hash, a Number, or another Object.

In Action Cable, our Channel object is the subscriber and our application (controllers, models, jobs, ...) is the publisher.



To create a stream we can use either *"stream_from"* which creates a PubSub channel using a String identifier that we pass to it *(e.g. "chat:1337")*. Or *"stream_for"* which accepts an object *(e.g. "Chat.find(1337)")*. *"stream_for"* internally generates a String identifier for the given object and calls *"stream_from"* with it.

```
#!/usr/bin/ruby
class ChatChannel < ApplicationCable::Channel
  def subscribe
    @chat_room = current_person
          .chat_rooms
          .find_by(name: params[:room_name])

        reject unless @chat_room

    # creates a PubSub channel to which we can publish messages to from
anywhere
```

```
    stream_for @chat_room
    # the above is basicaly the same as
    # stream_from "chat_room:#{@chat_room.id}"
  end
end
```

Remember how I earlier said that the server will confirm a subscription only in some cases? **Well, it will only confirm subscriptions which open a stream.** If a subscription doesn't open a stream the server just won't respond to a subscribe message.

Now that we have a stream we can publish messages to it from anywhere using *"broadcast_to"* which accepts two arguments - the object we are broadcasting to, and the messages to broadcast.

```
#!/usr/bin/ruby
ChatChannel.broacast_to(
  ChatRoom.find(params[:id]), # The same record that we gave to stream_for
  "Hello, Zagreb!" # The message I want to send
)
```

If you used *"stream_from"* to create your stream, then you have to use *"ActionCable.server.broadcast"* instead of *"broadcast_to"*. It also expects two arguments - the stream we are broadcasting to, and the message we are broadcasting.

```
#!/usr/bin/ruby
ActionCable.server.broadcast("chat_room:123", "Hello, Zagreb!")
```

You can also send a message directly from the Channel object using the *"transmit"* method.

```
#!/usr/bin/ruby
class ChatChannel < ApplicationCable::Channel
  def subscribe
    @chat_room = current_person
          .chat_rooms
          .find_by(name: params[:room_name])

        reject unless @chat_room

    transmit "Welcome back, #{current_person.first_name}!"
  end
end
```

Ok, but what happens when we broadcast or transmit a message?

Well, it's wrapped in a hash containing two keys - *"identifier"* and *"message"*. The identifier holds the same value that was sent in the original *"subscribe"* message. While the message

holds whatever you are sending to the client.

```
{
  "identifier": {
    "channel": "ChatChannel",
    "room_name": "Ruby Zagreb"
  },
  "message": "Hello, Zagreb!"
}
```

Then that Hash is turned into JSON and sent via the WebSocket.

When the client gets that message, it can figure out for which of its subscriptions it's for based on the *"identifier"*, and then it can process the *"message"* however it likes.

In the browser

Now we have sent a message from the server, but how can we respond to it in a browser?

Action Cable ships with an official JavaScript client that enables you to connect to a server via Action Cable, subscribe to any channel, and receive messages. Just like its server counterpart, it hides the protocol from you and allows you to focus only on the messages.

To connect to a server you have to create a *consumer*, which is a wrapper around a WebSocket connection to our server.

You can create a consumer that will connect to "/cable" without any extra params just by calling *"createConsumer"*.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()
```

If you want to pass extra params to the server's Connection object, e.g. for authentication, you'll have to pass the exact URL with params and all to *"createConsumer"*.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

// Fetches the user's auth token from a meta tag in the DOM
const token = document.querySelector("meta[name=authe_token]")?.content

// Takes the current URL,
// changes it's path to "/ws",
// and adds "?token=#{token}" as the query params.
// The result looks something like "https://example.com/ws?token=123"
```

```
const webSocketURL = new URL(window.location.href)
webSocketURL.pathname = "/ws" // usually it's "/cable"
webSocketURL.search = `?token=${token}`

const consumer = createConsumer(webSocketURL.toString())
```

Now that you have a consumer, you can subscribe with it to any channel you'd like. To subscribe, you have to create a subscription with an identifier.

If you remember from before, an identifier has to have a *"channel"* field, but can also have any additional fields you want and you'll have access to these fields as *"params"* in your Channel object.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()

// Subscribe to the ChatChannel
// and pass { room_name: "Ruby Zagreb" } as params
consumer.subscriptions.create(
  { channel: "ChatChannel", room_name: "Ruby Zagreb" }
)
```

How do I process an incoming message with this?

Well, [when your subscriptions receives a message it will try to call a *"received"* method of the subscription object](#) to process the message.

You can create a received method on the subscription yourself, like so

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()

// Subscribe to the ChatChannel
// and pass { room_name: "Ruby Zagreb" } as params
const subscription = consumer.subscriptions.create(
  { channel: "ChatChannel", room_name: "Ruby Zagreb" }
)

subscription.received = function(message) {
  document
    .querySelector("#messages")
    ?.insertAdjacentHTML("beforeend", `<div>${message}</div>`)
}
```

Though this approach has a problem - if you get a message immediately as you subscribe, but before you create your received method, you'll miss that message.

Action Cable allows you to extend the Subscription object as you create it to avoid this problem. You can pass a second argument while creating a subscription. That argument has to be an object with which the newly created Subscription object will be extended. Extending an object in JavaScript is similar to including a mixin in a class in Ruby.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()

// Subscribe to the ChatChannel
// and pass { room_name: "Ruby Zagreb" } as params
const subscription = consumer.subscriptions.create(
  { channel: "ChatChannel", room_name: "Ruby Zagreb" },
  {
    received(message) {
          this.appendMessage(message)
    }

    appendMessage(message) {
          this.messageContainer()?.insertAdjacentHTML(
                "beforeend",
                `<div>${message}</div>`
          )
    }

    messageContainer() {
          document.querySelector("#messages")
    }
  }
)
```

But there are other events that you can process in your Subscription object besides receiving a message.

You can process initialization events, connect and disconnect events, as well as subscription rejections. Each event requires you to define a method to process it. Initialization requires an *"initialized"* method, rejection a *"rejected"* method, connect and disconnect a *"connected"* and *"disconnected"* method.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()
```

```
const subscription = consumer.subscriptions.create(
  { channel: "ChatChannel", room_name: "Ruby Zagreb" },
  {
    // Called right after the Subscription object is created
    initialized() {
      console.log(`A subscription to ${this.identifier} was created`)
    }

    // Called if the subscription was rejected by the server
        rejected() {
          console.log(`The server rejected the subscription to
${this.identifier}`)
        }

    // Called when the subscription gets confirmed by the server
    connected(data) {
          // The data object has a single property `reconnected`
          // which indicates if this was a resubscribe after a disconnect
      console.log(`The server confirmed the subscription to
${this.identifier}! Reconnected: ${data.reconnected}`)
    }

    // Called when the WebSocket closes
        disconnected(data) {
          // The data object has a single propert `willAttemptReconnect`
          // which indicates if a reconnect attempt will be made or not
          console.log(`WebSocket to ${this.consumer.url} closed! Will
attempt reconnect: ${data.willAttemptReconnect}`)
        }

    // Called when a message is sent from the server
    received(message) {
          console.log(`Received message: ${message}`)
    }
  }
)
```

There are also two actions that you can perform from your subscription - send messages to the server, and unsubscribe.

To unsubscribe just call *"unsubscribe"*.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()

const subscription = consumer.subscriptions.create(
```

```
  { channel: "ChatChannel", room_name: "Ruby Zagreb" },
  {
    received(message) {
      console.log(`Received message: ${message}`)

      if (message.toLowerCase() === "avada kedavra") this.unsubscribe()
    }
  }
)
```

And to send a message call the *"send"* method with the message you want to send.

```
#!/usr/bin/node
import { createConsumer } from "@rails/actioncable"

const consumer = createConsumer()

const subscription = consumer.subscriptions.create(
  { channel: "ChatChannel", room_name: "Ruby Zagreb" }
)

subscription.send({ action: "post_message", content: "Hello, Zagreb!" })
```

Thousands of connections, one server

Now you know how to connect to Action Cable and send messages back and forth. But if you have ever tuned a Rails application you probably know that there is a thread pool that determines how many simulations requests your server can process.

Does that mean that there is a maximum number of connections that Action Cable can handle?

Well, no. As long as your server has memory available it will be able to accept and serve more Action Cable connections. Granted, it will process them slower and slower, but it will work.

How does that work?

Puma, currently the most popular Ruby/Rack application server, is a multi-threaded server. This means that when you make a request to it, your request will be assigned to one thread that will process it and generate a response.

But, you don't want to have too many threads. Each thread running on your server gets a very short slice of time to run its code on the processor. The more threads you have, the more any one thread has to wait to get to run its code on the processor.

So the smarter thing to do is to have a preset number of threads - a thread pool - that can process requests. If a request comes in, and all threads in the pool are busy, the request goes into a queue. It will wait there until a thread becomes available. That's why you want to take as little time as possible to generate a response for a request. The more a request takes to process, the higher the chance that some requests will end up in the queue which means that your app feels slow.

But Action Cable turns requests into WebSockets, which can stay open for days, how don't we run out of threads in the pool?

That's the interesting part. **Action Cable processes only the initial HTTP request in Puma's thread pool.** After you get a 101 response from the server the request is "hijacked" and put into an event loop. **So Puma's thread pool determines the maximum number of WebSocket connections that can be opened at once.**

Hijack? Event loop? What?

Rails doesn't integrate directly with an application server. Instead it implements Rack's protocol through which it gets requests from, and returns responses to, the application server. This enables applications and frameworks like Rails to work with any application server like Puma, WebBrick or Falcon.

Rack is great for request-response interactions like HTTP, but a WebSocket isn't request-response like. It's a bi-directional stream of data.

To support WebSockets and similar stream-like protocols, Rack offers a way to read and write bytes directly to a connection initiated by a request - it's called hijacking the socket.

When you hijack a Rack request you get it's underlying socket. You can read and write bytes to the sockets freely. But this means that it's your responsibility to implement whatever protocol you want over that socket.

Action Cable hijacks the socket and passes it to Faye's WebSocket Driver library which implements the WebSocket protocol and it puts that socket into an event loop.

An event loop is a pattern for responding to events. You start an infinite loop in which you wait for some event, when it occurs you process it and the loop starts again.

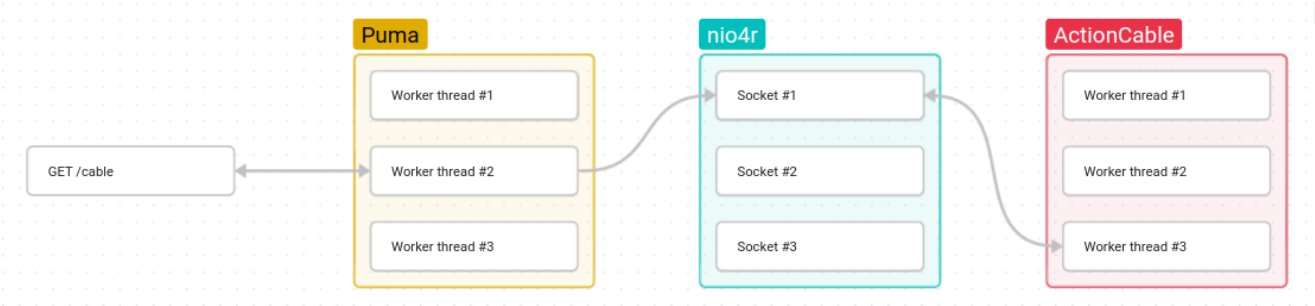
In Action Cable's case it's waiting for bytes to read from the socket it hijacked. When they are available, it reads them and passes them on to the WebSocket driver.

That's easy enough for one socket. But to wait for any of an infinite number of sockets to have some bytes ready to be read would require an infinite number of threads - one for each socket.

To avoid spawning a thread for each WebSocket, Action Cable uses nio4r which notifies it when a socket has some bytes ready to be read, without spawning any threads. It does so

using different functions of the server's operating system's kernel - such as select, epoll and kqueue.

Once it's notified that a socket is ready, it reads its bytes, parses them into a message, and then passes the message to the Connection object, which then puts in in a thread pool to be processed - called the worker pool - to process that message.



Another thread pool?

Yes. **This worker pool allows Action Cable to process multiple incoming messages simultaneously.**

You can tweak the size of this worker pool by setting *"config.action_cable.worker_pool_size"* to the number of threads you'd like to have in that pool.

The more threads in this pool the higher the number of incoming messages that you can process simultaneously. More threads also means that your latency will go up (as one threads will have to wait for longer to get access to the processor), your database connection count will go up (as more threads can access the database simultaneously), and your memory consumption will go up as more messages and objects are in memory at the same time.

But there is one more thread pool in Action Cable - the event loop thread pool.

This one is used to dispatch events like sending heart beat messages, subscribing to and unsubscribing from PubSub channels created by stream_from, attaching and detaching hijacked sockets, and triggering periodic timers (to which I'll get to in just a bit).

This thread pool has a fixed size as it's intended to be an event dispatch of sorts where you just schedule a task which the pool only trigger, the execution of the task is mostly done in the worker pool.

Things I wish I knew right away

There are a few things that I learned about Action Cable that I wish I knew earlier.

You can trigger actions and send messages periodically, like every few seconds. This is extremely useful if you have some state that you want to synchronize periodically, or some hose keeping you want to do.

In your Channel object you can call a *"periodically"* class method, give it a method name or a block, and an interval. It will then run that method/block for you with whatever interval you specified.

```
#!/usr/bin/ruby
class StockTickerChannel < ApplicationCable::Channel
  # this will send a message to the client every 2 seconds
  # as long as they are subscribed
  periodically every: 2.seconds do
    transmit value: @stock.value, timestamp: Time.now.to_i
  end

  def subscribe
    @stock = Stock.find_by(symbol: params[:symbol])

        reject unless @stock
  end
end
```

Some error trackers won't catch errors from your Channel objects unless you explicitly send them. You can do that in your Connection using *"rescue_from"*, just like you would in a Controller.

```
#!/usr/bin/ruby
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    rescue_from Exception do |error|
      MyErrorTracker.capture_exception(error)
    end

    # ...
  end
end
```

There is a callback on the Connection object for when an action gets invoked on your Channel object. You can register such a callback using *"before_command"*, *"after_command"* and "around_command", which is extremely useful if you use Current attributes.

```
#!/usr/bin/ruby
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    around_command do
          Current.set(person: current_person) { yield }
    end
```

```
    # ...
  end
end
```

There are callbacks on the Channel object for when someone subscribes or unsubscribes. There is "*before_subscribe*", *"after_subscribe"*, *"around_subscribe"*, and "*before_unsubscribe*", *"after_unsubscribe"*, *"around_unsubscribe"*. These is useful for implementing common behavior through inheritance or mixins without having to call "super" form the *"subscribe"* or *"unsubscribe"* method.

```
#!/usr/bin/ruby
module AppearanceTrackable
  extend ActiveSupport::Concern

  included do
        after_subscribe unless: :subscription_rejected? do
          Current.person.came_online!
        end

        after_unsubscribe do
          Current.person.went_offline!
    end
  end
end

class ChatChannel < ApplicationCable::Channel
  include AppearanceTrackable

  def subscribed
    @chat_room = current_person
          .chat_rooms
          .find_by(name: params[:room_name])

        reject unless @chat_room

    stream_for @chat_room
  end
end
```

You probably want to bump Puma's worker timeout if you are debugging your Connection object. Puma will kill any worker that doesn't generate a response within 60 seconds.

This can be annoying if you are trying to debug something like how you authenticate Connections with *"debugger"*, *"binding.irb"* or *"binding.pry"*.

But you can raise that timeout using the *"worker_timeout"* method in *"puma.rb"*.

```
#!/usr/bin/ruby
# config/puma.rb
require File.expand_path("../config/environment", File.dirname(__FILE__))

# ...

# Kill a worker thread if it didn't generate a responsw in 8 hours
worker_timeout 8 * 3600 if ENV.fetch("RAILS_ENV", "development") ==
"development"
```

You have to tweak both Puma's thread and Action Cable's worker counts. Puma's thread count controls how many new WebSocket connections can be created simultaneously. Action Cable's worker pool size determines how many WebSocket messages you can process simultaneously. Both pools will create database connections!

```
#!/usr/bin/ruby
# config/puma.rb
max_threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }
min_threads_count = ENV.fetch("RAILS_MIN_THREADS") { max_threads_count }
threads min_threads_count, max_threads_count

# config/application.rb
config.action_cable.worker_pool_size = ENV.fetch("RAILS_MAX_THREADS") { 5
}
```

In development, sometimes the server can behave a bit wonky. I'm not a 100% sure what's going on but it seems like code reloading can cause rogue threads with stale code. If that happens just run "touch tmp/restart.txt" to quickly restart the server and reset everything.

When a client looses it's Internet connection it will take Action Cable up to 20 min to notice that, close the connection, and trigger callbacks. I wrote about that in my previous article. This can cause you headaches if you want to allow only a certain number of connections per client.

You can use the official JS client outside of the browser, like in Node or Bun. The official client doesn't use the browser's WebSocket object directly. It exports an *"adapters"* object which has a property called *"WebSocket"* which holds the object that will be used to establish WebSocket connections. So you can drop-in your own WebSocket object and use it.

You can remotely disconnect any client. There is a *"ActionCable::RemoteConnections"* class that acts like an Active Record relation. You can query it using its *"where"* method to get a connection. The where method searches for a connection by its *"identified_by"* fields. If

it finds such a connection it gives you a proxy for that connection on which you can only call *"disconnect"* which then disconnects the client.

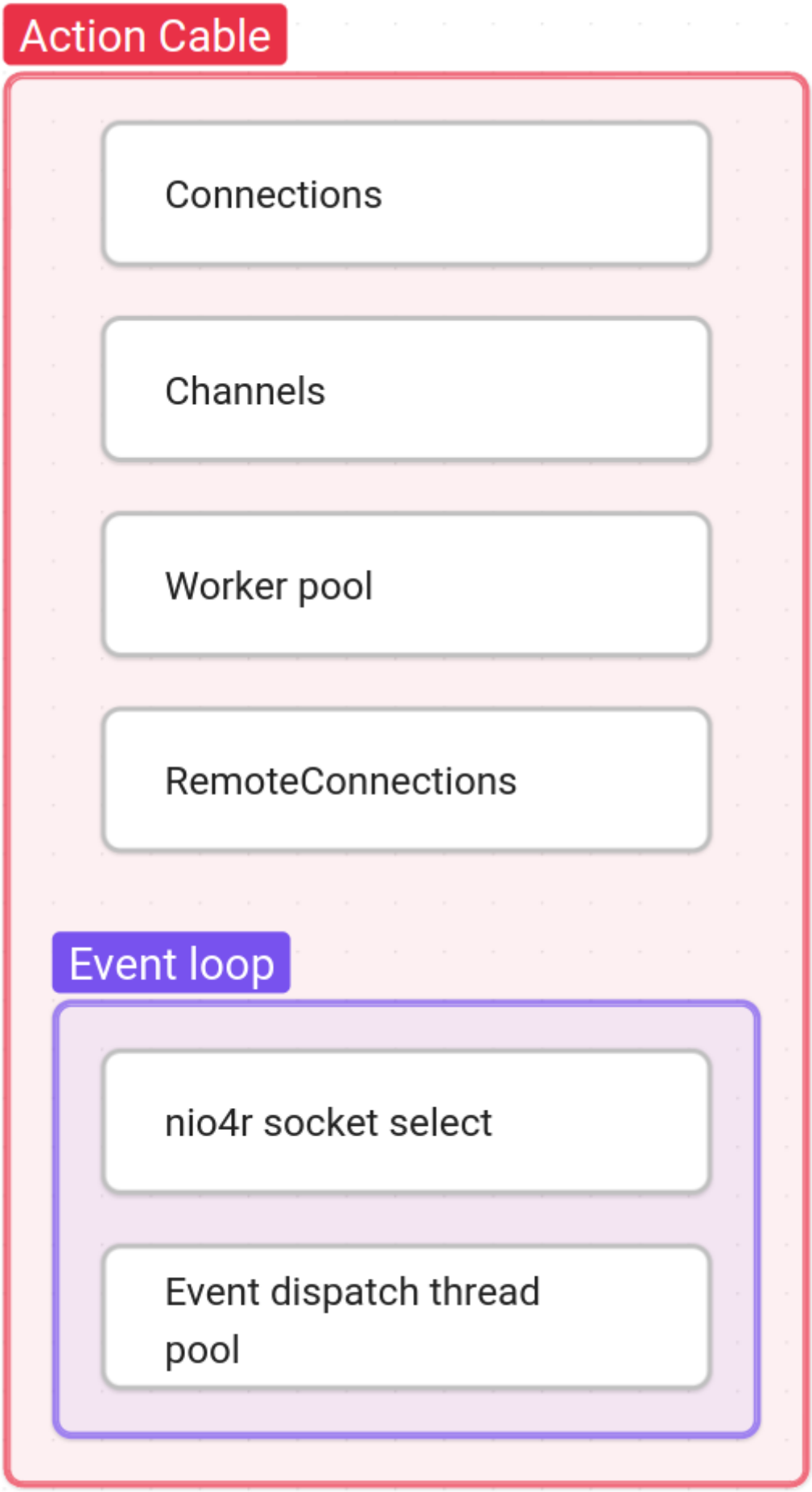
```
#!/usr/bin/ruby
class Person < Applicationrecord
  def ban!
    update!(banned: true)

    ActionCable::RemoteConnections
      .where(current_person: self)
      &.disconnect(reconnect: false)
  end
end
```

There are multiple PubSub adapters available besides Redis. You can use Postgres instead of Redis as a backend (it has some limitations). And you can provide your own if you need.

```
# config/cable.yml
production:
  adapter: postgres
```

Final thought

Action Cable is an amazing piece of software. It can seem complex, but that's because it does a lot. When you look at each piece individually the complexity goes away.


Action Cable
Connections
Channels
Worker pool
RemoteConnections
Event loop
nio4r socket select
Event dispatch thread pool