Action Cable Overview

In this guide, you will learn how Action Cable works and how to use WebSockets to incorporate real-time features into your Rails application.

After reading this guide, you will know:

- What Action Cable is and its integration backend and frontend
- How to set up Action Cable
- How to set up channels
- Deployment and Architecture setup for running Action Cable

What is Action Cable?

Action Cable seamlessly integrates WebSockets with the rest of your Rails application. It allows for real-time features to be written in Ruby in the same style and form as the rest of your Rails application, while still being performant and scalable. It's a full-stack offering that provides both a client-side JavaScript framework and a server-side Ruby framework. You have access to your entire domain model written with Active Record or your ORM of choice.

Terminology

Action Cable uses WebSockets instead of the HTTP request-response protocol. Both Action Cable and WebSockets introduce some less familiar terminology:

Connections

Connections form the foundation of the client-server relationship. A single Action Cable server can handle multiple connection instances. It has one connection instance per WebSocket connection. A single user may have multiple WebSockets open to your application if they use multiple browser tabs or devices.

Consumers

The client of a WebSocket connection is called the *consumer*. In Action Cable, the consumer is created by the client-side JavaScript framework.

Channels

Each consumer can, in turn, subscribe to multiple *channels*. Each channel encapsulates a logical unit of work, similar to what a controller does in a typical MVC setup. For example, you could have a `ChatChannel` and an `AppearancesChannel`, and a consumer could be subscribed to either or both of these channels. At the very least, a consumer should be subscribed to one channel.

Subscribers

When the consumer is subscribed to a channel, they act as a *subscriber*. The connection between the subscriber and the channel is, surprise-surprise, called a subscription. A consumer can act as a subscriber to a given channel any number of times. For example, a consumer could subscribe to multiple chat rooms at the same time. (And remember that a physical user may have multiple consumers, one per tab/device open to your connection).

Pub/Sub

Pub/Sub or Publish-Subscribe refers to a message queue paradigm whereby senders of information (publishers), send data to an abstract class of recipients (subscribers), without specifying individual recipients. Action Cable uses this approach to communicate between the server and many clients.

Broadcastings

A broadcasting is a pub/sub link where anything transmitted by the broadcaster is sent directly to the channel subscribers who are streaming that named broadcasting. Each channel can be streaming zero or more broadcastings.

Server-Side Components

Connections

For every WebSocket accepted by the server, a connection object is instantiated. This object becomes the parent of all the *channel subscriptions* that are created from thereon. The connection itself does not deal with any specific application logic beyond authentication and authorization. The client of a WebSocket connection is called the connection *consumer*. An individual user will create one consumer-connection pair per browser tab, window, or device they have open.

Connections are instances of `ApplicationCable::Connection`, which extends `ActionCable::Connection::Base`. In `ApplicationCable::Connection`, you authorize the incoming connection and proceed to establish it if the user can be identified.

Connection Setup

```ruby
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
    end

    private
      def find_verified_user
        if verified_user = User.find_by(id: cookies.encrypted[:user_id])
          verified_user
        else
          reject_unauthorized_connection
        end
      end
  end
end
```

Here `identified_by` designates a connection identifier that can be used to find the specific connection later. Note that anything marked as an identifier will automatically create a delegate by the same name on any channel instances created off the connection.

This example relies on the fact that you will already have handled authentication of the user somewhere else in your application, and that a successful authentication sets an encrypted cookie with the user ID.

The cookie is then automatically sent to the connection instance when a new connection is attempted, and you use that to set the `current_user`. By identifying the connection by this same current user, you're also ensuring that you can later retrieve all open connections by a given user (and potentially disconnect them all if the user is deleted or unauthorized).

If your authentication approach includes using a session, you use cookie store for the session, your session cookie is named `_session` and the user ID key is `user_id` you can use this approach:

```
verified_user = User.find_by(id: cookies.encrypted['_session']['user_id'])
```

Exception Handling

By default, unhandled exceptions are caught and logged to Rails' logger. If you would like to globally intercept these exceptions and report them to an external bug tracking service, for example, you can do so with `rescue_from`:

```
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    rescue_from StandardError, with: :report_error

    private

    def report_error(e)
      SomeExternalBugtrackingService.notify(e)
    end
  end
end
```

Channels

A *channel* encapsulates a logical unit of work, similar to what a controller does in a typical MVC setup. By default, Rails creates a parent `ApplicationCable::Channel` class (which extends `ActionCable::Channel::Base`) for encapsulating shared logic between your channels.

Parent Channel Setup

```
# app/channels/application_cable/channel.rb
module ApplicationCable
  class Channel < ActionCable::Channel::Base
  end
end
```

Then you would create your own channel classes. For example, you could have a `ChatChannel` and an `AppearanceChannel`:

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
end
```

```
# app/channels/appearance_channel.rb
class AppearanceChannel < ApplicationCable::Channel
end
```

A consumer could then be subscribed to either or both of these channels.

Subscriptions

Consumers subscribe to channels, acting as *subscribers*. Their connection is called a *subscription*. Produced messages are then routed to these channel subscriptions based on an identifier sent by the channel consumer.

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  # Called when the consumer has successfully
  # become a subscriber to this channel.
  def subscribed
  end
end
```

Exception Handling

As with `ApplicationCable::Connection`, you can also use `rescue_from` on a specific channel to handle raised exceptions:

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  rescue_from 'MyError', with: :deliver_error_message

  private

  def deliver_error_message(e)
    broadcast_to(...)
  end
end
```

Client-Side Components

Connections

Consumers require an instance of the connection on their side. This can be established using the following JavaScript, which is generated by default by Rails:

Connect Consumer

```
// app/javascript/channels/consumer.js
// Action Cable provides the framework to deal with WebSockets in Rails.
// You can generate new channels where WebSocket features live using the `bin/rails
generate channel` command.

import { createConsumer } from "@rails/actioncable"
```

```
export default createConsumer()
```

This will ready a consumer that'll connect against `/cable` on your server by default. The connection won't be established until you've also specified at least one subscription you're interested in having.

The consumer can optionally take an argument that specifies the URL to connect to. This can be a string or a function that returns a string that will be called when the WebSocket is opened.

```
// Specify a different URL to connect to
createConsumer('https://ws.example.com/cable')

// Use a function to dynamically generate the URL
createConsumer(getWebSocketURL)

function getWebSocketURL() {
  const token = localStorage.get('auth-token')
  return `https://ws.example.com/cable?token=${token}`
}
```

Subscriber

A consumer becomes a subscriber by creating a subscription to a given channel:

```
// app/javascript/channels/chat_channel.js
import consumer from "./consumer"

consumer.subscriptions.create({ channel: "ChatChannel", room: "Best Room" })

// app/javascript/channels/appearance_channel.js
import consumer from "./consumer"

consumer.subscriptions.create({ channel: "AppearanceChannel" })
```

While this creates the subscription, the functionality needed to respond to received data will be described later on.

A consumer can act as a subscriber to a given channel any number of times. For example, a consumer could subscribe to multiple chat rooms at the same time:

```
// app/javascript/channels/chat_channel.js
import consumer from "./consumer"

consumer.subscriptions.create({ channel: "ChatChannel", room: "1st Room" })
consumer.subscriptions.create({ channel: "ChatChannel", room: "2nd Room" })
```

Client-Server Interactions

Streams

Streams provide the mechanism by which channels route published content (broadcasts) to their subscribers. For example, the following code uses `stream_from` to subscribe to the broadcasting named `chat_Best Room` when the value of the `:room` parameter is `"Best Room"`:

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end
end
```

Then, elsewhere in your Rails application, you can broadcast to such a room by calling `broadcast`:

```
ActionCable.server.broadcast("chat_Best Room", { body: "This Room is Best Room." })
```

If you have a stream that is related to a model, then the broadcasting name can be generated from the channel and model. For example, the following code uses `stream_for` to subscribe to a broadcasting like `comments:Z2lkOi8vVGVzdEFwcC9Qb3N0LzE`, where `Z2lkOi8vVGVzdEFwcC9Qb3N0LzE` is the GlobalID of the Post model.

```
class CommentsChannel < ApplicationCable::Channel
  def subscribed
    post = Post.find(params[:id])
    stream_for post
  end
end
```

You can then broadcast to this channel by calling `broadcast_to`:

```
CommentsChannel.broadcast_to(@post, @comment)
```

Broadcastings

A *broadcasting* is a pub/sub link where anything transmitted by a publisher is routed directly to the channel subscribers who are streaming that named broadcasting. Each channel can be streaming zero or more broadcastings.

Broadcastings are purely an online queue and time-dependent. If a consumer is not streaming (subscribed to a given channel), they'll not get the broadcast should they connect later.

Subscriptions

When a consumer is subscribed to a channel, they act as a subscriber. This connection is called a subscription. Incoming messages are then routed to these channel subscriptions based on an identifier sent by the cable consumer.

```
// app/javascript/channels/chat_channel.js
import consumer from "./consumer"

consumer.subscriptions.create({ channel: "ChatChannel", room: "Best Room" }, {
  received(data) {
    this.appendLine(data)
  },

  appendLine(data) {
    const html = this.createLine(data)
    const element = document.querySelector("[data-chat-room='Best Room']")
    element.insertAdjacentHTML("beforeend", html)
  },

  createLine(data) {
    return `
      <article class="chat-line">
        <span class="speaker">${data["sent_by"]}</span>
        <span class="body">${data["body"]}</span>
      </article>
    `
  }
})
```

Passing Parameters to Channels

You can pass parameters from the client-side to the server-side when creating a subscription. For example:

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end
end
```

An object passed as the first argument to `subscriptions.create` becomes the params hash in the cable channel. The keyword `channel` is required:

```
// app/javascript/channels/chat_channel.js
import consumer from "./consumer"

consumer.subscriptions.create({ channel: "ChatChannel", room: "Best Room" }, {
  received(data) {
    this.appendLine(data)
  },

  appendLine(data) {
    const html = this.createLine(data)
    const element = document.querySelector("[data-chat-room='Best Room']")
    element.insertAdjacentHTML("beforeend", html)
  },

  createLine(data) {
    return `
      <article class="chat-line">
```

```
        <span class="speaker">${data["sent_by"]}</span>
        <span class="body">${data["body"]}</span>
      </article>
    `
  }
})
```

```
# Somewhere in your app this is called, perhaps
# from a NewCommentJob.
ActionCable.server.broadcast(
  "chat_#{room}",
  {
    sent_by: 'Paul',
    body: 'This is a cool chat app.'
  }
)
```

Rebroadcasting a Message

A common use case is to *rebroadcast* a message sent by one client to any other connected clients.

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end

  def receive(data)
    ActionCable.server.broadcast("chat_#{params[:room]}", data)
  end
end
```

```
// app/javascript/channels/chat_channel.js
import consumer from "./consumer"

const chatChannel = consumer.subscriptions.create({ channel: "ChatChannel", room:
"Best Room" }, {
  received(data) {
    // data => { sent_by: "Paul", body: "This is a cool chat app." }
  }
}

chatChannel.send({ sent_by: "Paul", body: "This is a cool chat app." })
```

The rebroadcast will be received by all connected clients, *including* the client that sent the message. Note that params are the same as they were when you subscribed to the channel.

Full-Stack Examples

The following setup steps are common to both examples:

1. Set up your connection.
2. Set up your parent channel.
3. Connect your consumer.

Example 1: User Appearances

Here's a simple example of a channel that tracks whether a user is online or not and what page they're on. (This is useful for creating presence features like showing a green dot next to a username if they're online).

Create the server-side appearance channel:

```
# app/channels/appearance_channel.rb
class AppearanceChannel < ApplicationCable::Channel
  def subscribed
    current_user.appear
  end

  def unsubscribed
    current_user.disappear
  end

  def appear(data)
    current_user.appear(on: data['appearing_on'])
  end

  def away
    current_user.away
  end
end
```

When a subscription is initiated the `subscribed` callback gets fired, and we take that opportunity to say "the current user has indeed appeared". That appear/disappear API could be backed by Redis, a database, or whatever else.

Create the client-side appearance channel subscription:

```
// app/javascript/channels/appearance_channel.js
import consumer from "./consumer"

consumer.subscriptions.create("AppearanceChannel", {
  // Called once when the subscription is created.
  initialized() {
    this.update = this.update.bind(this)
  },

  // Called when the subscription is ready for use on the server.
  connected() {
    this.install()
    this.update()
  },
  // Called when the WebSocket connection is closed.
  disconnected() {
    this.uninstall()
  },

  // Called when the subscription is rejected by the server.
  rejected() {
    this.uninstall()
  },

  update() {
    this.documentIsActive ? this.appear() : this.away()
  },

  appear() {
    // Calls `AppearanceChannel#appear(data)` on the server.
    this.perform("appear", { appearing_on: this.appearingOn })
  },

  away() {
    // Calls `AppearanceChannel#away` on the server.
    this.perform("away")
  },

  install() {
    window.addEventListener("focus", this.update)
    window.addEventListener("blur", this.update)
    document.addEventListener("turbolinks:load", this.update)
    document.addEventListener("visibilitychange", this.update)
  },

  uninstall() {
    window.removeEventListener("focus", this.update)
    window.removeEventListener("blur", this.update)
    document.removeEventListener("turbolinks:load", this.update)
    document.removeEventListener("visibilitychange", this.update)
  },

  get documentIsActive() {
    return document.visibilityState === "visible" && document.hasFocus()
  },

  get appearingOn() {
    const element = document.querySelector("[data-appearing-on]")
    return element ? element.getAttribute("data-appearing-on") : null
  }
})
```

Client-Server Interaction

1. **Client** connects to the **Server** via `App.cable = ActionCable.createConsumer("ws://cable.example.com")` . (`cable.js`). The **Server** identifies this connection by `current_user` .
2. **Client** subscribes to the appearance channel via `consumer.subscriptions.create({ channel: "AppearanceChannel" })` . (`appearance_channel.js`)
3. **Server** recognizes a new subscription has been initiated for the appearance channel and runs its `subscribed` callback, calling the `appear` method on `current_user` . (`appearance_channel.rb`)
4. **Client** recognizes that a subscription has been established and calls `connected` (`appearance_channel.js`), which in turn calls `install` and `appear` . `appear` calls `AppearanceChannel#appear(data)` on the server, and supplies a data hash of `{ appearing_on: this.appearingOn }` . This is possible because the server-side channel instance automatically exposes all public methods declared on the class (minus the callbacks), so that these can be reached as remote procedure calls via a subscription's `perform` method.
5. **Server** receives the request for the `appear` action on the appearance channel for the connection identified by `current_user` (`appearance_channel.rb`). **Server** retrieves the data with the `:appearing_on` key from the data hash and sets it as the value for the `:on` key being passed to `current_user.appear` .

Example 2: Receiving New Web Notifications

The appearance example was all about exposing server functionality to client-side invocation over the WebSocket connection. But the great thing about WebSockets is that it's a two-way street. So, now, let's show an example where the server invokes an action on the client.

This is a web notification channel that allows you to trigger client-side web notifications when you broadcast to the relevant streams:

Create the server-side web notifications channel:

```ruby
# app/channels/web_notifications_channel.rb
class WebNotificationsChannel < ApplicationCable::Channel
  def subscribed
    stream_for current_user
  end
end
```

Create the client-side web notifications channel subscription:

```js
// app/javascript/channels/web_notifications_channel.js
// Client-side which assumes you've already requested
// the right to send web notifications.
import consumer from "./consumer"

consumer.subscriptions.create("WebNotificationsChannel", {
  received(data) {
    new Notification(data["title"], { body: data["body"] })
  }
})
```

Broadcast content to a web notification channel instance from elsewhere in your application:

```
# Somewhere in your app this is called, perhaps from a NewCommentJob
WebNotificationsChannel.broadcast_to(
  current_user,
  title: 'New things!',
  body: 'All the news fit to print'
)
```

The `WebNotificationsChannel.broadcast_to` call places a message in the current subscription adapter's pubsub queue under a separate broadcasting name for each user. For a user with an ID of 1, the broadcasting name would be `web_notifications:1`.

The channel has been instructed to stream everything that arrives at `web_notifications:1` directly to the client by invoking the `received` callback. The data passed as an argument is the hash sent as the second parameter to the server-side broadcast call, JSON encoded for the trip across the wire and unpacked for the data argument arriving as `received`.

More Complete Examples

See the rails/actioncable-examples repository for a full example of how to set up Action Cable in a Rails app and adding channels.

Configuration

Action Cable has two required configurations: a subscription adapter and allowed request origins.

Subscription Adapter

By default, Action Cable looks for a configuration file in `config/cable.yml`. The file must specify an adapter for each Rails environment. See the Dependencies section for additional information on adapters.

```
development:
  adapter: async

test:
  adapter: test

production:
  adapter: redis
  url: redis://10.10.3.153:6381
  channel_prefix: appname_production
```

Adapter Configuration

Below is a list of the subscription adapters available for end-users.

Async Adapter

The async adapter is intended for development/testing and should not be used in production.

Redis Adapter

The Redis adapter requires users to provide a URL pointing to the Redis server. Additionally, a `channel_prefix` may be provided to avoid channel name collisions when using the same Redis server for multiple applications. See the Redis PubSub documentation for more details.

The Redis adapter also supports SSL/TLS connections. The required SSL/TLS parameters can be passed in `ssl_params` key in the configuration YAML file.

```
production:
  adapter: redis
  url: rediss://10.10.3.153:tls_port
  channel_prefix: appname_production
  ssl_params: {
    ca_file: "/path/to/ca.crt"
  }
```

The options given to `ssl_params` are passed directly to the `OpenSSL::SSL::SSLContext#set_params` method and can be any valid attribute of the SSL context. Please refer to the OpenSSL::SSL::SSLContext documentation for other available attributes.

If you are using self-signed certificates for redis adapter behind a firewall and opt to skip certificate check, then the ssl `verify_mode` should be set as `OpenSSL::SSL::VERIFY_NONE`.

WARNING: It is not recommended to use `VERIFY_NONE` in production unless you absolutely understand the security implications. In order to set this option for the Redis adapter, the config should be `ssl_params: { verify_mode: <%= OpenSSL::SSL::VERIFY_NONE %> }`.

PostgreSQL Adapter

The PostgreSQL adapter uses Active Record's connection pool, and thus the application's `config/database.yml` database configuration, for its connection. This may change in the future. #27214

Allowed Request Origins

Action Cable will only accept requests from specified origins, which are passed to the server config as an array. The origins can be instances of strings or regular expressions, against which a check for the match will be performed.

```
config.action_cable.allowed_request_origins = ['https://rubyonrails.com',
%r{http://ruby.*}]
```

To disable and allow requests from any origin:

```
config.action_cable.disable_request_forgery_protection = true
```

By default, Action Cable allows all requests from localhost:3000 when running in the development environment.

Consumer Configuration

To configure the URL, add a call to `action_cable_meta_tag` in your HTML layout HEAD. This uses a URL or path typically set via `config.action_cable.url` in the environment configuration files.

Worker Pool Configuration

The worker pool is used to run connection callbacks and channel actions in isolation from the server's main thread. Action Cable allows the application to configure the number of simultaneously processed threads in the worker pool.

```
config.action_cable.worker_pool_size = 4
```

Also, note that your server must provide at least the same number of database connections as you have workers. The default worker pool size is set to 4, so that means you have to make at least 4 database connections available. You can change that in `config/database.yml` through the `pool` attribute.

Client-side logging

Client-side logging is disabled by default. You can enable this by setting the `ActionCable.logger.enabled` to true.

```
import * as ActionCable from '@rails/actioncable'

ActionCable.logger.enabled = true
```

Other Configurations

The other common option to configure is the log tags applied to the per-connection logger. Here's an example that uses the user account id if available, else "no-account" while tagging:

```
config.action_cable.log_tags = [
  -> request { request.env['user_account_id'] || "no-account" },
  :action_cable,
  -> request { request.uuid }
]
```

For a full list of all configuration options, see the `ActionCable::Server::Configuration` class.

Running Standalone Cable Servers

In App

Action Cable can run alongside your Rails application. For example, to listen for WebSocket requests on `/websocket`, specify that path to [`config.action_cable.mount_path`]:

```
# config/application.rb
class Application < Rails::Application
  config.action_cable.mount_path = '/websocket'
end
```

You can use `ActionCable.createConsumer()` to connect to the cable server if `action_cable_meta_tag` is invoked in the layout. Otherwise, A path is specified as first argument to `createConsumer` (e.g. `ActionCable.createConsumer("/websocket")`).

For every instance of your server you create, and for every worker your server spawns, you will also have a new instance of Action Cable, but the Redis or PostgreSQL adapter keeps messages synced across connections.

Standalone

The cable servers can be separated from your normal application server. It's still a Rack application, but it is its own Rack application. The recommended basic setup is as follows:

```
# cable/config.ru
require_relative "../config/environment"
Rails.application.eager_load!

run ActionCable.server
```

Then you start the server using a binstub in `bin/cable` ala:

```
#!/bin/bash
bundle exec puma -p 28080 cable/config.ru
```

The above will start a cable server on port 28080.

Notes

The WebSocket server doesn't have access to the session, but it has access to the cookies. This can be used when you need to handle authentication. You can see one way of doing that with Devise in this article.

Dependencies

Action Cable provides a subscription adapter interface to process its pubsub internals. By default, asynchronous, inline, PostgreSQL, and Redis adapters are included. The default adapter in new Rails applications is the asynchronous (`async`) adapter.

The Ruby side of things is built on top of websocket-driver, nio4r, and concurrent-ruby.

Deployment

Action Cable is powered by a combination of WebSockets and threads. Both the framework plumbing and user-specified channel work are handled internally by utilizing Ruby's native thread support. This means you can use all your existing Rails models with no problem, as long as you haven't committed any thread-safety sins.

The Action Cable server implements the Rack socket hijacking API, thereby allowing the use of a multi-threaded pattern for managing connections internally, irrespective of whether the application server is multi-threaded or not.

Accordingly, Action Cable works with popular servers like Unicorn, Puma, and Passenger.

Testing

You can find detailed instructions on how to test your Action Cable functionality in the testing guide.