Hotrails

← Back to the list of chapters

Real-time updates with Turbo Streams

Published on February 02, 2022

In this chapter, we will learn how to broadcast Turbo Stream templates with Action Cable to make real-time updates on a web page.

Real-time updates with Turbo Streams and Action Cable

The Turbo Stream format allows, in combination with Action Cable, to make real-time updates to our web pages with just a few lines of code. Real-world applications are, for example, group chats, notifications, or email services.

Let's take the example of an email service. When we receive a new email, we don't want to refresh our page manually to see what's new. Instead, **we want our email feed to be updated in real-time with fresh content without having to do anything**. We want the changes that happen server-side to be pushed to our browser without any manual action on our side.

Implementing this real-time behavior in Rails became easier when Action Cable was released in version 5 of the framework. **The part of Turbo Rails we will talk about in this chapter is built on top of Action Cable**. It is now even easier to implement real-time behavior in Rails, and it requires very little code.

What we will build in this chapter

Let's imagine that multiple users will be using our quote editor. They would like to see real-time updates of what their colleagues are working on:

On the `Quotes#index` page:

- Every time a colleague creates a quote, we want it to be prepended to our quotes list in real-time.
- Every time a colleague updates a quote, we want to see this update reflected on our quotes list in real-time.
- Every time a colleague deletes a quote, we want it to disappear from our quotes list in real-time.

Even if this example feels a bit cumbersome, it will enable us to learn how to use Turbo Streams to make real-time updates to our `Quotes#index` page. What we will learn in the next chapter is also valid for notifications, emails, or any `ActiveRecord` model.

Broadcasting created quotes with Turbo Streams

Let's transform the `Quotes#index` page into a real-time page. Every time a colleague creates a quote, we want the created quote to appear on our `Quotes#index` page in real-time without manually refreshing the page.

To do this, we have to tell our `Quote` model to broadcast the HTML of the created quote to the users of our quote editor right after it was created. Let's update our `Quote` model:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes", partial: "quot
end
```

Let's break down this line of code together. If it does not make sense now, it will become clearer later when we do some experiments in the browser.

First, we use an `after_create_commit` callback to instruct our Ruby on Rails application that the expression in the lambda should be executed **every time a new quote is inserted into the database**.

The second part of the expression in the lambda is more complex. It instructs our Ruby on Rails application that the *HTML of the created quote* should be broadcasted to users subscribed to the `"quotes"` stream and prepended to the DOM node with the id of `"quotes"`.

What does that mean exactly?

We will explain later how to subscribe to the `"quotes"` stream and receive the HTML in the browser, but for now, let's focus on what HTML is being generated.

As instructed, the `broadcast_prepend_to` method will render the `quotes/_quote.html.erb` partial in the Turbo Stream format with the action `prepend` and the target `"quotes"` as specified with the `target: "quotes"` option:

```
<turbo-stream action="prepend" target="quotes">
  <template>
    <turbo-frame id="quote_123">
      <!-- The HTML for the quote partial -->
    </turbo-frame>
  </template>
</turbo-stream>
```

If we remember what we learned about the Turbo Stream format in the previous chapter, we should notice this is **the same HTML** as the one that was generated in the `QuotesController#create` action to *prepend* the created quote to the list of quotes! When Turbo receives this kind of HTML, it is smart enough to interpret it and *prepend* the content of the `<template>` to the DOM node with id `"quotes"`.

The only difference is that the HTML is delivered via WebSocket this time instead of in response to an AJAX request!

Note: We want the created quote to be *prepended* to the DOM node with id `"quotes"` in this example. We could also like the new quote to be *appended* to the `"quotes"` target by using `broadcast_append_to` instead of `broadcast_prepend_to`.

For our users to subscribe to the `"quotes"` stream, we need to specify it in the `Quotes#index` view. Let's add a single line of code at the top of our view:

```
<%# app/views/quotes/index.html.erb %>

<%= turbo_stream_from "quotes" %>

<%# All the previous HTML markup %>
```

The HTML generated by the `turbo_stream_from` helper looks like this:

```
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="very-long-string"
>
</turbo-cable-stream-source>
```

The `turbo_stream_from` helper generates a custom element used in the Turbo JavaScript library to subscribe users to the channel named in the `channel` attribute and, more specifically, to the stream named in the `signed-stream-name` attribute.

The `Turbo::StreamsChannel` inside the `channel` attribute is the name of the Action Cable channel. Turbo Rails always uses this channel, so this attribute is always the same.

The `signed-stream-name` attribute is the *signed* version of the `"quotes"` string we passed as an argument. It is *signed* to prevent malicious users from tampering with it and receiving HTML from streams they should not have access to. We will explain this more in-depth in the next chapter about security. For now, we only need to know that we can decode this string and read its original value: `"quotes"`.

All of the users on the `Quotes#index` page are now subscribed to the `Turbo::StreamsChannel` and waiting for broadcastings to the `"quotes"` stream. Every time a new quote is inserted in the database, those users will receive HTML in the Turbo Stream format, and Turbo will prepend the markup for the created quote to the list of quotes.

Now let's test that everything works as expected. There are two ways to test manually that our code works as expected, and we will explore both of them.

Testing Turbo Streams in the console

In this chapter, every time we make a change to the `Quote` model and want to test in the console, we have to restart the rails console before performing the test. We might see some unexpected results otherwise.

Note: Before you start your tests in the console, you have to ensure that Redis is properly configured in your application.

In development, your `config/cable.yml` should look like this:

```
# config/cable.yml

development:
  adapter: redis
  url: redis://localhost:6379/1

# All the rest of the file
```

If that's the case, you can skip the rest of this note.

Otherwise, **you first have to install Redis** that is used by Action Cable on your computer and then run the `bin/rails turbo:install` command. It should update the `config/cable.yml` file in development to the configuration shown above. Once that's the case, you can continue reading the tutorial!

To perform our test, let's open the `Quotes#index` page in the browser. The first way to test that everything is wired correctly is to open the rails console

and create a new quote:

```
Quote.create!(name: "Broadcasted quote")
```

What do we see in the console logs when doing this? The first thing is the quote creation itself:

```
TRANSACTION (0.1ms)  begin transaction
Quote Create (0.4ms)  INSERT INTO "quotes" ("name", "created_at", "updat
TRANSACTION (0.8ms)  commit transaction
```

As we can see, the quote was created in the database, and the transaction was committed. However, there is something new happening. We should see the following lines in the console:

```
Rendered quotes/_quote.html.erb (Duration: 0.5ms | Allocations: 285)
[ActionCable] Broadcasting to quotes: "<turbo-stream action=\"prepend\"
```

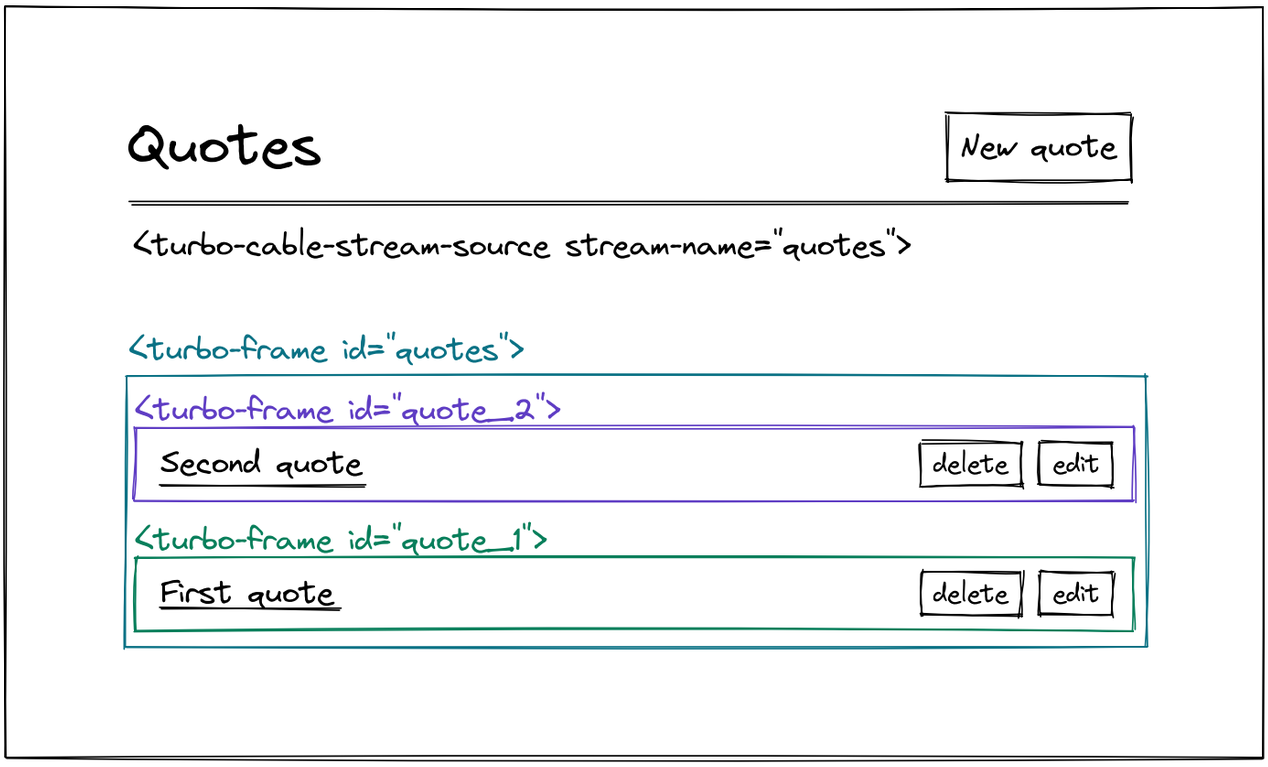
This is quite a lot of text, but there are very interesting parts.

The first thing we notice is that this HTML was broadcasted via `ActionCable` to a stream named `"quotes"`. Thanks to the `turbo_stream_from "quotes"` line we added to our `Quotes#index` view previously, we are subscribed to the stream and, thus, will receive the HTML that is broadcasted.

The second thing we notice is that the broadcasted HTML is in the Turbo Stream format. It instructs Turbo to `"prepend"` the content of the `<template>` to the target `"quotes"`. Indeed that is what we instructed the `Quote` model to do!

The third and last thing we should notice is that the HTML contained in the `<template>` was generated by the `quotes/_quote.html.erb` partial for the quote that was just created. When Turbo receives this template in the frontend, it will append it to the DOM node with the id of `"quotes"`.

Let's sketch this behavior. Our `Quotes#index` page now looks like this:



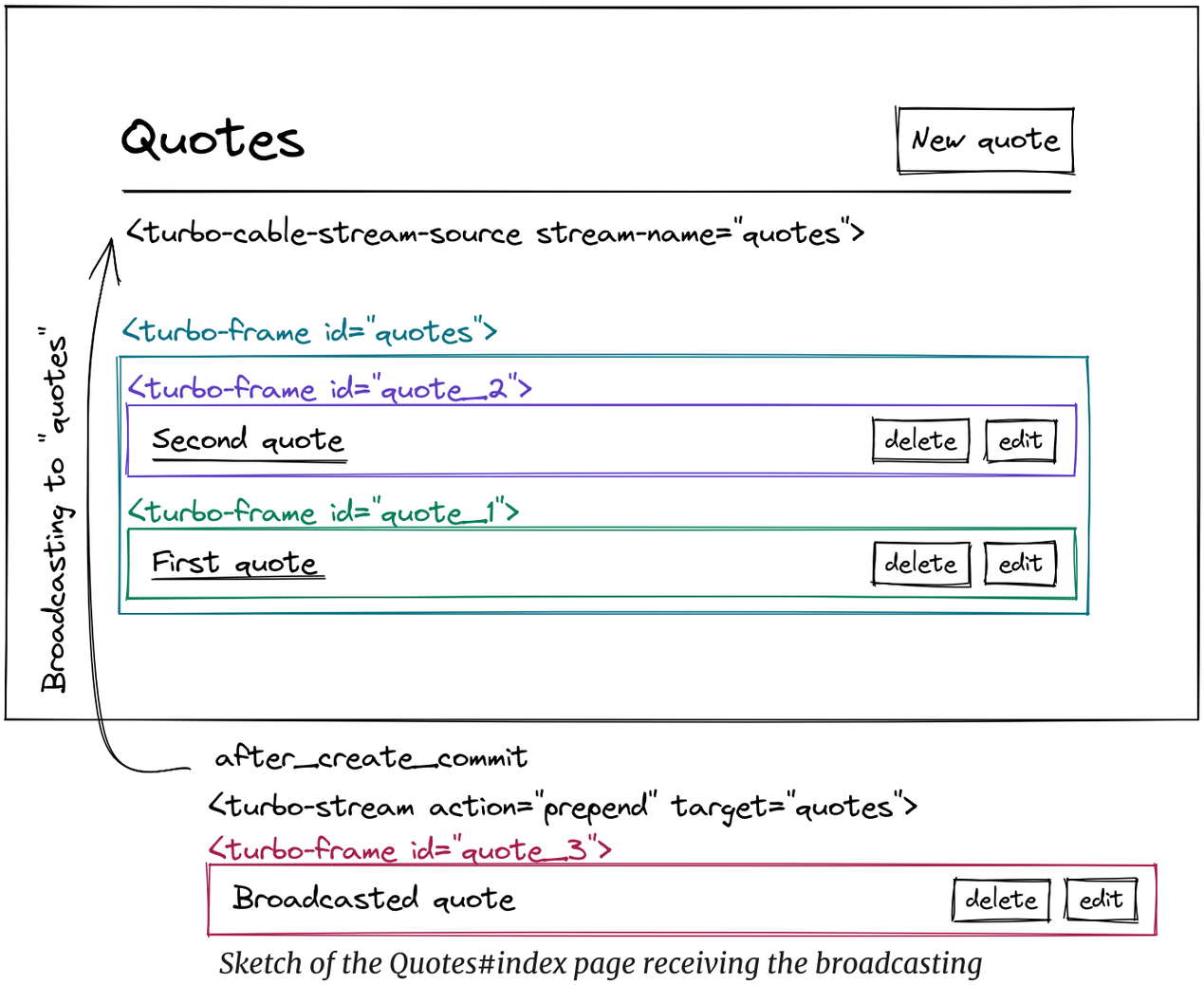
Sketch of the Quotes#index page

Now let's imagine a colleague creates a new quote.

Thanks to the `after_create_commit` callback, the `broadcasts_prepend_to` method is called when the created quote is added to the database.

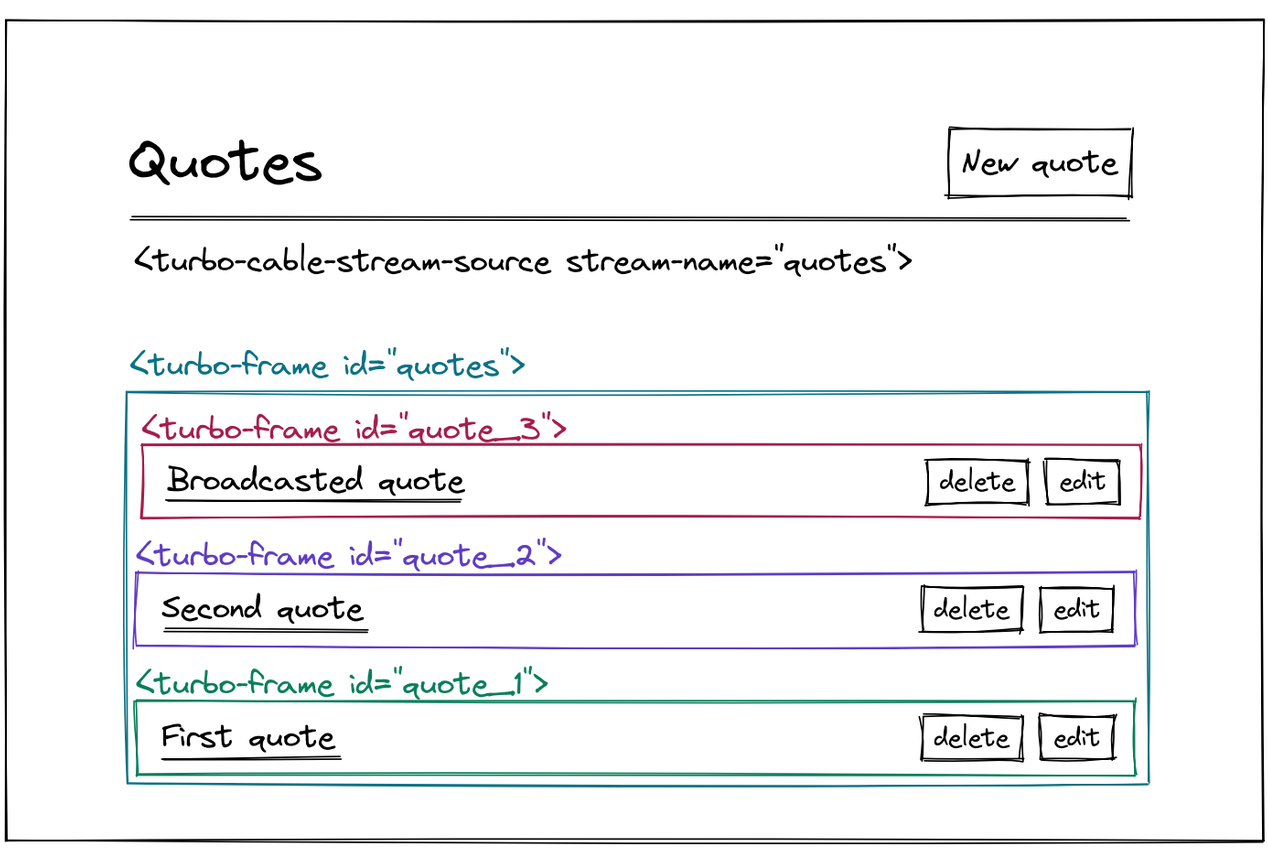
We subscribed to those broadcastings on the `Quotes#index` page using the `turbo_stream_from` method.

Those two lines of code are described in the sketch below:



Sketch of the Quotes#index page receiving the broadcasting

In our browser, we should see that a quote named "Broadcasted quote" has been prepended to the list of quotes in real-time:



Sketch of the Quotes#index page with the created quote prepended to the list

Thanks to Turbo Rails, which is built on top of Action Cable, those changes can be reflected in real-time to all users subscribed to the right channel with the

right stream name. We didn't have to refresh the page to see the change! We turned our application into a real-time one with just two lines of code!

Testing Turbo Streams with two browser windows

Another way to test that everything is working as expected is to open two browser windows on the `Quotes#index` page and put them side by side. In one of the two windows, let's create a quote. We should see that the change is immediately reflected in the other window without refreshing the page.

As we can see, our application is reactive as expected!

Turbo Streams conventions and syntactic sugar

We can reduce the amount of code we wrote in this first part of the chapter in the `Quote` model:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes", partial: "quot
end
```

As we can see above, we specify the target name to be `"quotes"` thanks to the `target: "quotes"` option. By default, the target option will be equal to `model_name.plural`, which is equal to `"quotes"` in the context of our `Quote` model. Thanks to this convention, we can remove the `target: "quotes"` option:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code
```

```
  after_create_commit -> { broadcast_prepend_to "quotes", partial: "quot
end
```

There are two other conventions we can use to shorten our code. Under the hood, Turbo has a default value for both the `partial` and the `locals` option.

The `partial` default value is equal to calling `to_partial_path` on an instance of the model, which by default in Rails for our `Quote` model is equal to `"quotes/quote"`.

The `locals` default value is equal to `{ model_name.element.to_sym => self }` which, in in the context of our `Quote` model, is equal to `{ quote: self }`.

These are precisely the values that we passed as options. Thus, the following code is equivalent to what we had before:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes" }
end
```

Using Ruby on Rails conventions, we made our application real-time with two (very short) lines of code!

Now that we understand how Turbo Streams work, it will be straightforward to finalize our real-time CRUD on the `Quote` model.

Broadcasting quote updates with Turbo Streams

Now that our real-time quote creation feature is working, let's add the same feature for quote updates.

Let's instruct our model to also broadcast updates on quotes:

```ruby
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes" }
  after_update_commit -> { broadcast_replace_to "quotes" }
end
```

That's it! It already works if we test in the console or with two browser windows! To understand how, let's test in the rails console:

```ruby
Quote.first.update!(name: "Update from console")
```

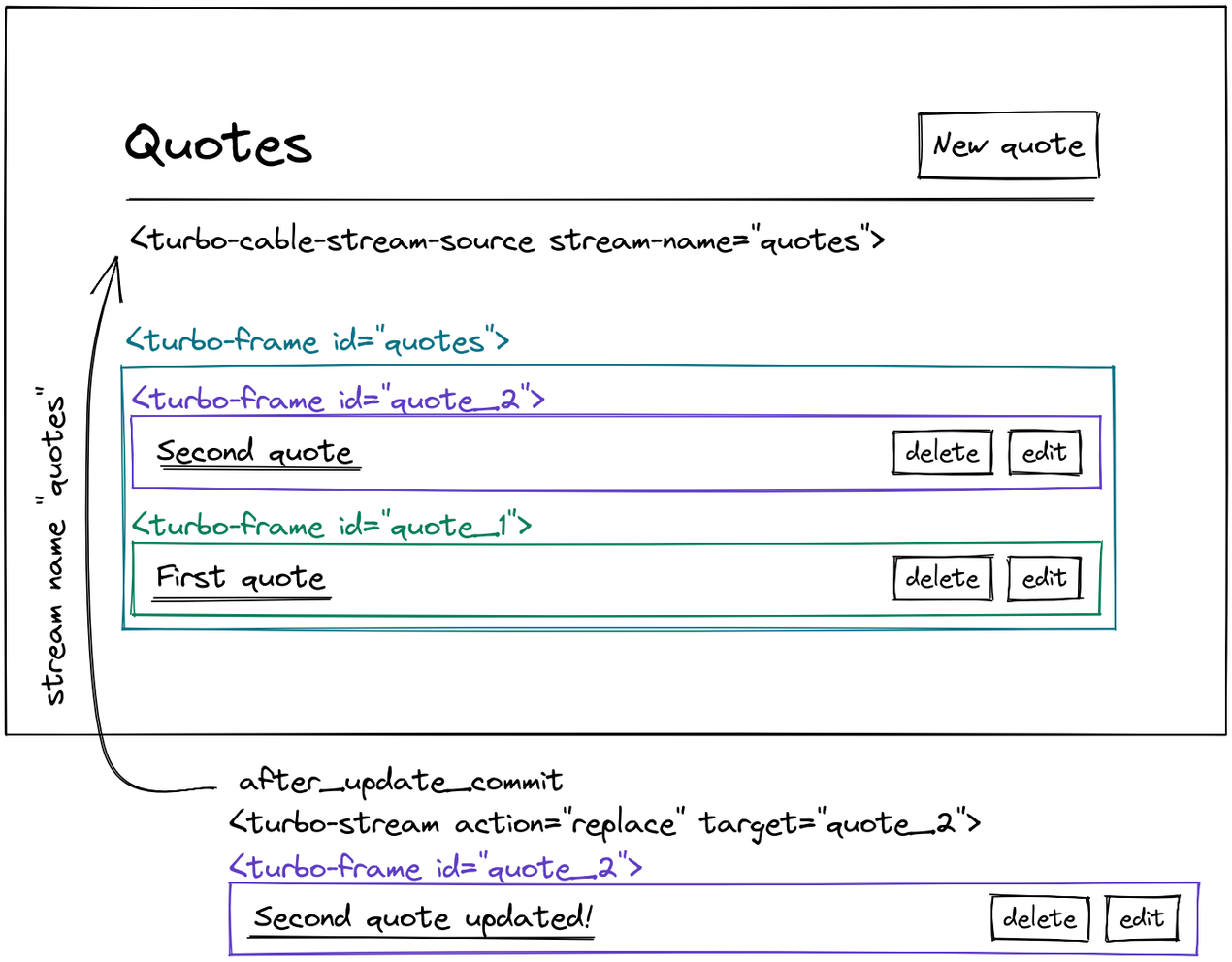
Just like before on quote creation, we notice in the console logs that the quote is updated in the database and that the transaction is committed:

```
Quote Load (0.3ms)  SELECT "quotes".* FROM "quotes" ORDER BY "quotes"."i
TRANSACTION (0.0ms)  begin transaction
Quote Update (0.3ms)  UPDATE "quotes" SET "name" = ?, "updated_at" = ? W
TRANSACTION (1.6ms)  commit transaction
```

When the transaction is committed, the `after_update_commit` callback on the `Quote` model is triggered and calls the `broadcast_replace_to` method:

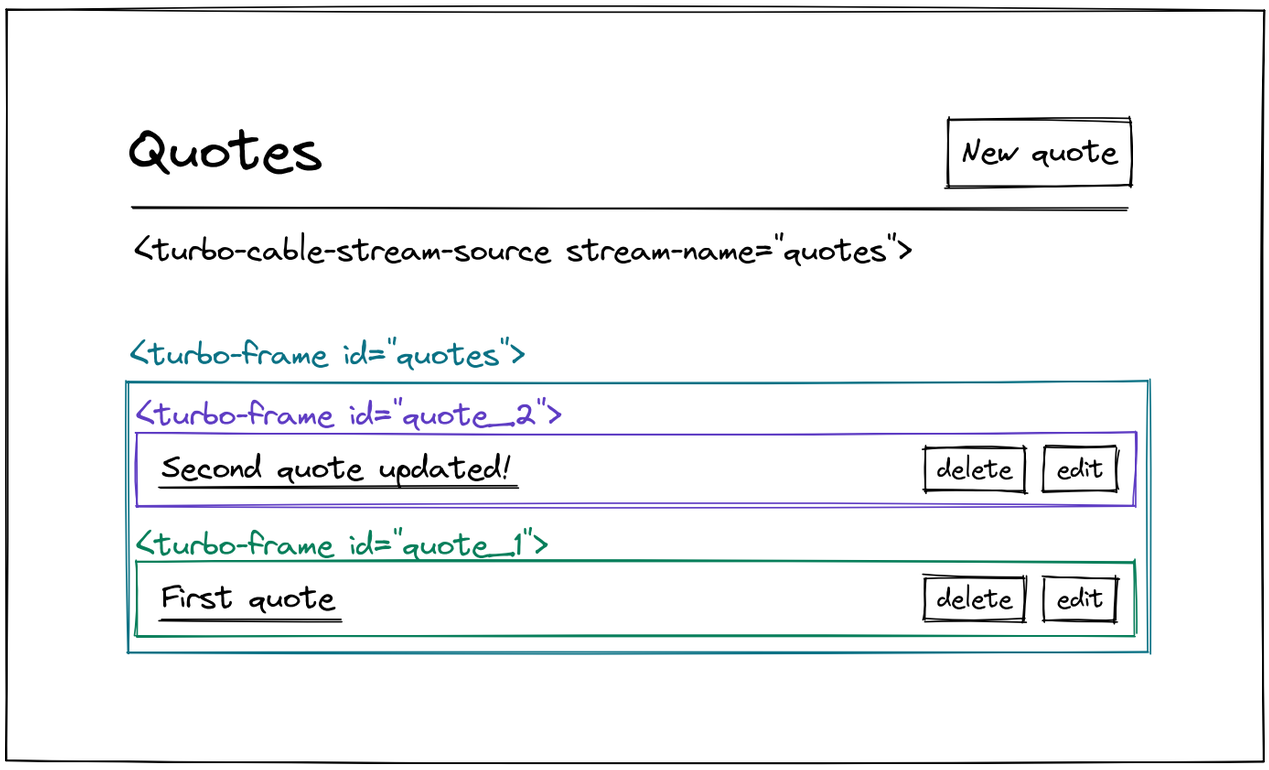
```
Rendered quotes/_quote.html.erb (Duration: 0.6ms | Allocations: 285)
[ActionCable] Broadcasting to quotes: "<turbo-stream action=\"replace\"
```

Like last time, we can see that the HTML of the `quotes/quote` partial is broadcasted to the `"quotes"` stream. The main difference is that this time, the action is `"replace"` and not `"prepend"`, and that the `target` DOM node is the quote card with the id of `"quote_908005754"` where `"908005754"` is the id of the updated quote:



Sketch of the Quotes#index page receiving the broadcasting

Turbo intercepts the received HTML, and the quote card is replaced:



Sketch of the Quotes#index page with the updated quote

The last feature we want to implement is to make our application real-time when a user deletes a quote. That's what we will do in the next section!

Broadcasting quote deletion with Turbo Streams

Let's instruct our `Quote` model to broadcast changes when a quote was deleted from the database. Just like before, this is done by using a callback on the model:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes" }
  after_update_commit -> { broadcast_replace_to "quotes" }
  after_destroy_commit -> { broadcast_remove_to "quotes" }
end
```

Let's test the feature immediately in the rails console to ensure it works as expected. Let's make sure we have a quote in our local database that we can destroy and run the following command:

```
Quote.last.destroy!
```

As we can see in the browser, it works as expected! Let's analyze the logs to understand why.

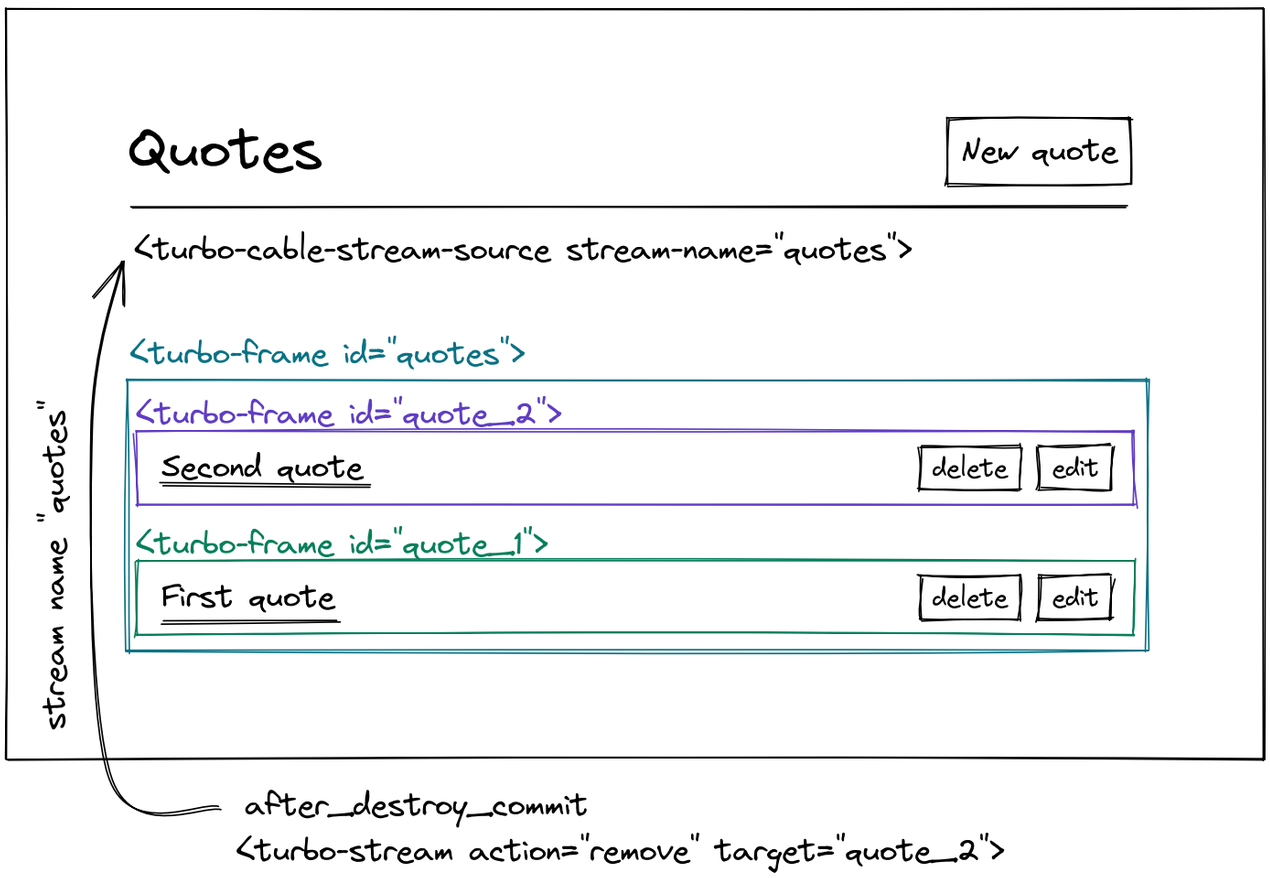
```
Quote Load (0.3ms)  SELECT "quotes".* FROM "quotes" ORDER BY "quotes"."i
TRANSACTION (0.1ms)  begin transaction
Quote Destroy (0.4ms)  DELETE FROM "quotes" WHERE "quotes"."id" = ?  [["
TRANSACTION (1.4ms)  commit transaction
```

In this first part of the logs, we can see that the last quote is retrieved from the database and then destroyed in a transaction. When the transaction ends, the `after_destroy_commit` callback is triggered from our `Quote` model and calls the `broadcast_remove_to` method:

```
[ActionCable] Broadcasting to quotes: "<turbo-stream action=\"remove\" t
```

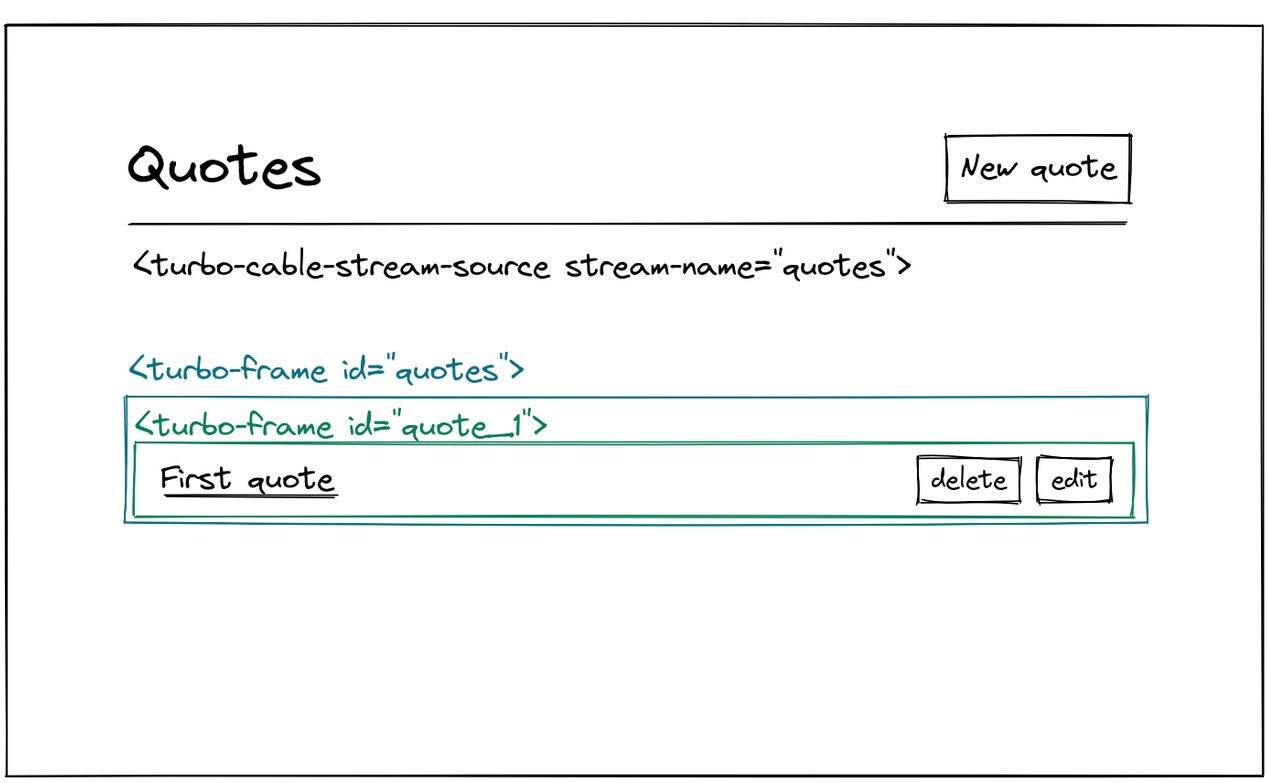
Some HTML is broadcasted by the `"quotes"` channel to our users. This time, this HTML only instructs Turbo to remove the DOM node of id

`"quote_908005754"`, where `"908005754"` is the database id of the quote that was just deleted:



Sketch of the Quotes#index page receiving the broadcasting

As a result, the quote disappears from the `Quotes#index` page for all the users that are on the `Quotes#index` page:



Sketch of the Quotes#index page with the quote removed from the list

We just finalized our real-time CRUD on the `Quote` model. This is exciting! Before we finish this chapter and go to the next one, we need to talk about

performance.

Making broadcasting asynchronous with ActiveJob

Our `Quote` model currently looks like this:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_to "quotes" }
  after_update_commit -> { broadcast_replace_to "quotes" }
  after_destroy_commit -> { broadcast_remove_to "quotes" }
end
```

It is possible to improve the performance of this code by making the broadcasting part asynchronous using background jobs. To do this, we only have to update the content of our callbacks to use their asynchronous equivalents:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  after_create_commit -> { broadcast_prepend_later_to "quotes" }
  after_update_commit -> { broadcast_replace_later_to "quotes" }
  after_destroy_commit -> { broadcast_remove_to "quotes" }
end
```

Note: The `broadcast_remove_later_to` method does not exist because as the quote gets deleted from the database, it would be impossible for a background job to retrieve this quote in the database later to perform the job.

To see the difference, let's open our rails console and create a new quote:

```
Quote.create!(name: "Asynchronous quote")
```

Looking at the logs closely, we will see that the result is the same as when we created a quote, but the broadcasting part happens asynchronously. As we can see, a `Turbo::Streams::ActionBroadcastJob` is enqueued with all the necessary data to perform the broadcasting later.

```
Enqueued Turbo::Streams::ActionBroadcastJob (Job ID: 1eecd0c8-53fd-43ed-
```

The job is then performed to render the HTML of the `quotes/_quote.html.erb` partial just like before:

```
Performing Turbo::Streams::ActionBroadcastJob (Job ID: 1eecd0c8-53fd-43e
```

Broadcasting Turbo Streams asynchronously is the preferred method for performance reasons.

More syntactic sugar

If we had multiple real-time models, we would notice that the callbacks that we write in those models are very similar. Ruby on Rails is a fantastic framework, so there is some syntactic sugar to avoid writing those three callbacks all the time. Let's replace them with an equivalent and shorter version in our `Quote` model:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  # after_create_commit -> { broadcast_prepend_later_to "quotes" }
  # after_update_commit -> { broadcast_replace_later_to "quotes" }
  # after_destroy_commit -> { broadcast_remove_to "quotes" }
  # Those three callbacks are equivalent to the following single line
  broadcasts_to ->(quote) { "quotes" }, inserts_by: :prepend
end
```

The three callbacks are equivalent to a single line of code. We will discuss why we need a lambda in the next chapter about Turbo Streams and security. For now, let's understand that this means that we want to broadcast quote creations, updates, and deletions to the `"quotes"` stream asynchronously.

Our final `Quote` model implementation looks like this:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

  broadcasts_to ->(quote) { "quotes" }, inserts_by: :prepend
end
```

Wrap up

Transforming our application into a real-time one only took two lines of code, thanks to Turbo Rails.

1. In our `Quote` model, we set three callbacks to broadcast creations, updates, and deletions to the `"quotes"` stream. Thanks to the `broadcasts_to` method, those three callbacks can be defined in one line.
2. In our `Quotes#index` view, we explicitly mentioned that we want to subscribe to the changes that are broadcasted to the `"quotes"` stream.

Turbo takes care of all the rest, making our Ruby on Rails application a joy to work with.

In the next chapter, we will cover Turbo Streams and security. We will discuss how security works with Turbo Streams to ensure we don't broadcast private data to the wrong users.

← previous next →

Subscribe to the newsletter

Github Twitter Newsletter

Made with remotely