← Volver a la lista de capítulos

Actualizaciones en tiempo real con Turbo Streams

Publicado el 02 de febrero de 2022

En este capítulo, aprenderemos cómo transmitir plantillas de Turbo Stream con Action Cable para realizar actualizaciones en tiempo real en una página web.

Actualizaciones en tiempo real con Turbo Streams y Action Cable

El formato Turbo Stream permite, en combinación con Action Cable, realizar actualizaciones en tiempo real de nuestras páginas web con tan solo unas líneas de código. Las aplicaciones del mundo real son, por ejemplo, los chats grupales, las notificaciones o los servicios de correo electrónico.

Tomemos el ejemplo de un servicio de correo electrónico. Cuando recibimos un nuevo correo electrónico, no queremos actualizar nuestra página manualmente para ver qué hay de nuevo. En cambio, **queremos que nuestro feed de correo electrónico se actualice en tiempo real con contenido nuevo sin tener que hacer nada** . Queremos que los cambios que se producen en el servidor se envíen a nuestro navegador sin ninguna acción manual de nuestra parte.

Implementar este comportamiento en tiempo real en Rails se hizo más fácil cuando se lanzó Action Cable en la versión 5 del framework. **La parte de Turbo**

Rails de la que hablaremos en este capítulo está construida sobre Action Cable. Ahora es aún más fácil implementar el comportamiento en tiempo real en Rails y requiere muy poco código.

Lo que construiremos en este capítulo

Imaginemos que varios usuarios utilizan nuestro editor de citas y desean ver actualizaciones en tiempo real de lo que están haciendo sus colegas:

En la `Quotes#index` página:

- Cada vez que un colega crea una cotización, queremos que se agregue a nuestra lista de cotizaciones en tiempo real.
- Cada vez que un colega actualiza una cotización, queremos ver esta actualización reflejada en nuestra lista de cotizaciones en tiempo real.
- Cada vez que un colega elimina una cita, queremos que desaparezca de nuestra lista de citas en tiempo real.

Aunque este ejemplo parezca un poco engorroso, nos permitirá aprender a utilizar Turbo Streams para realizar actualizaciones en tiempo real de nuestra `Quotes#index` página. Lo que aprenderemos en el siguiente capítulo también es válido para notificaciones, correos electrónicos o cualquier `ActiveRecord` modelo.

Transmisión de citas creadas con Turbo Streams

Transformemos la `Quotes#index` página en una página en tiempo real. Cada vez que un colega cree una cotización, queremos que la cotización creada aparezca en nuestra `Quotes#index` página en tiempo real sin tener que actualizar la página manualmente.

Para ello, tenemos que indicarle a nuestro `Quote` modelo que transmita el código HTML de la cita creada a los usuarios de nuestro editor de citas inmediatamente después de su creación. Actualicemos nuestro `Quote` modelo:

```
# app/models/quote.rb

class Quote < ApplicationRecord
```

```
  # All the previous code

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

Analicemos juntos esta línea de código. Si no tiene sentido ahora, se aclarará más adelante cuando hagamos algunos experimentos en el navegador.

Primero, usamos una `after_create_commit` devolución de llamada para indicar a nuestra aplicación Ruby on Rails que la expresión en lambda debe ejecutarse **cada vez que se inserta una nueva cita en la base de datos** .

La segunda parte de la expresión en lambda es más compleja. Indica a nuestra aplicación Ruby on Rails que el *HTML de la cita creada* debe transmitirse a los usuarios suscritos a la `"quotes"` transmisión y anteponerse al nodo DOM con el id de `"quotes"` .

¿Qué significa eso exactamente?

Más adelante explicaremos cómo suscribirse al `"quotes"` stream y recibir el HTML en el navegador, pero por ahora, centrémonos en qué HTML se está generando.

Según las instrucciones, el `broadcast_prepend_to` método representará el `quotes/_quote.html.erb` parcial en el formato Turbo Stream con la acción `prepend` y el objetivo `"quotes"` especificados con la `target:` `"quotes"` opción:

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

Si recordamos lo que aprendimos sobre el formato Turbo Stream en el capítulo anterior, deberíamos notar que este es **el mismo HTML** que el que se generó en la `QuotesController#create` acción para *anteponer* la cita creada a la lista de citas. Cuando Turbo recibe este tipo de HTML, es lo suficientemente inteligente

como para interpretarlo y *anteponer* el contenido del `<template>` al nodo DOM con id `"quotes"`.

La única diferencia es que esta vez el HTML se entrega a través de WebSocket en lugar de en respuesta a una solicitud AJAX.

Nota : Queremos que la cita creada se *anteponga* al nodo DOM con id `"quotes"` en este ejemplo. También podríamos querer que la nueva cita se *adjunte* al `"quotes"` destino utilizando `broadcast_append_to` en lugar de `broadcast_prepend_to`.

Para que nuestros usuarios se suscriban a la `"quotes"` transmisión, debemos especificarlo en la `Quotes#index` vista. Agreguemos una sola línea de código en la parte superior de nuestra vista:

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

<%= turbo_stream_from "quotes" %>

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

El HTML generado por el `turbo_stream_from` ayudante se ve así:

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

El `turbo_stream_from` ayudante genera un elemento personalizado que se utiliza en la biblioteca Turbo JavaScript para suscribir a los usuarios al canal nombrado en el `channel` atributo y, más específicamente, a la transmisión nombrada en el `signed-stream-name` atributo.

Dentro `Turbo::StreamsChannel` del `channel` atributo se encuentra el nombre del canal de Action Cable. Turbo Rails siempre utiliza este canal, por lo que este atributo es siempre el mismo.

El `signed-stream-name` atributo es la versión *firmada* `"quotes"` de la cadena que pasamos como argumento. Está *firmada* para evitar que usuarios malintencionados la manipulen y reciban HTML de transmisiones a las que no deberían tener acceso. Explicaremos esto con más profundidad en el próximo capítulo sobre seguridad. Por ahora, solo necesitamos saber que podemos decodificar esta cadena y leer su valor original: `"quotes"` .

Todos los usuarios de la `Quotes#index` página están ahora suscritos a la transmisión `Turbo::StreamsChannel` y esperan las transmisiones en `"quotes"` directo. Cada vez que se inserte una nueva cita en la base de datos, esos usuarios recibirán HTML en formato Turbo Stream y Turbo antepondrá el marcado de la cita creada a la lista de citas.

Ahora, probemos que todo funcione como se espera. Hay dos formas de probar manualmente que nuestro código funcione como se espera y las exploraremos.

Probando Turbo Streams en la consola

En este capítulo, cada vez que realizamos un cambio en el `Quote` modelo y queremos probar en la consola, tenemos que reiniciar la consola de Rails antes de realizar la prueba. De lo contrario, es posible que veamos algunos resultados inesperados .

Nota : antes de comenzar sus pruebas en la consola, debe asegurarse de que Redis esté configurado correctamente en su aplicación.

En desarrollo, `config/cable.yml` debería verse así:

```
# config/cable.yml

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

# All the rest of the file
```

Si ese es el caso, puedes omitir el resto de esta nota.

De lo contrario, **primero debe instalar Redis** , que es el que usa Action Cable, en su computadora y luego ejecutar el `bin/rails turbo:install` comando. Debería actualizar el `config/cable.yml` archivo en desarrollo a la configuración que se muestra arriba. Una vez que ese sea el caso, ¡puede continuar leyendo el tutorial!

Para realizar nuestra prueba, abramos la `Quotes#index` página en el navegador. La primera forma de comprobar que todo está conectado correctamente es abrir la consola de Rails y crear una nueva cita:

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

¿Qué vemos en los logs de la consola al hacer esto? Lo primero es la creación de la cita en sí:

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

Como podemos ver, la cotización se creó en la base de datos y la transacción se confirmó. Sin embargo, está sucediendo algo nuevo. Deberíamos ver las siguientes líneas en la consola:

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

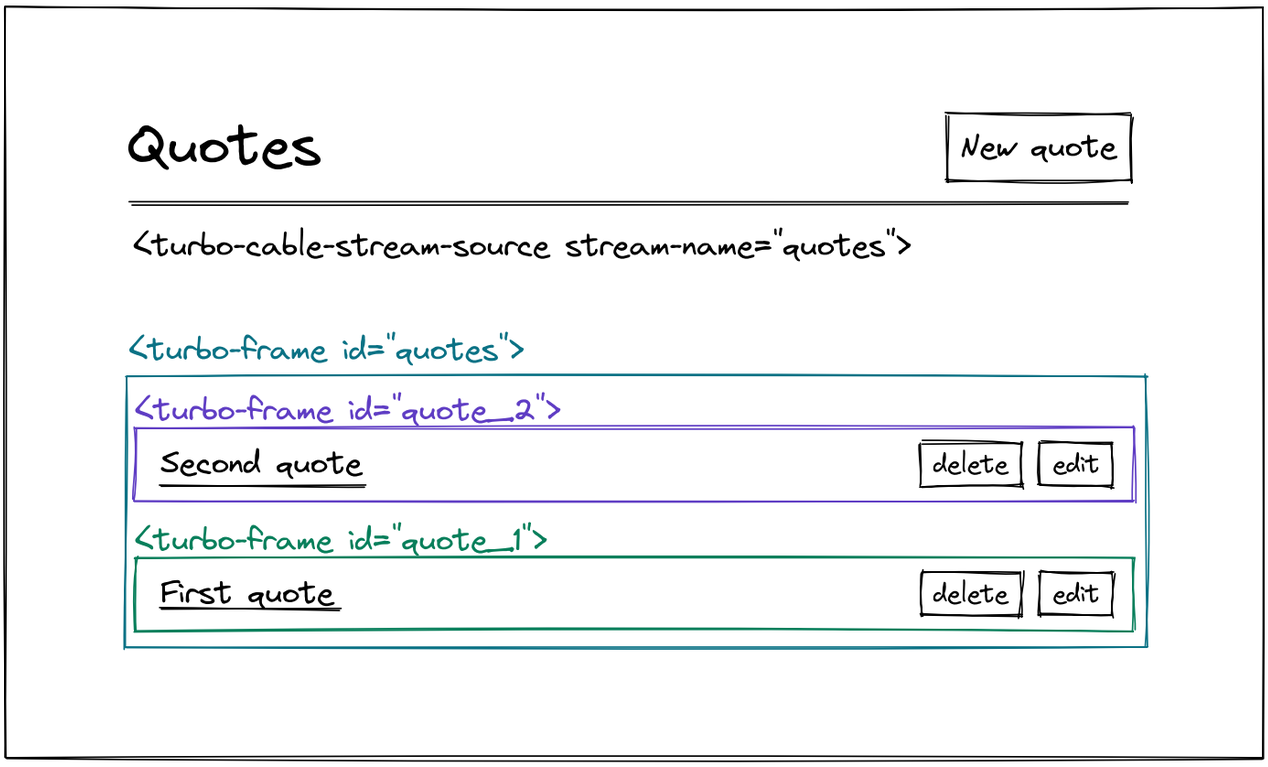
Es un texto bastante largo, pero hay partes muy interesantes.

Lo primero que notamos es que este HTML fue transmitido a través `ActionCable` de un stream llamado `"quotes"` . Gracias a la `turbo_stream_from "quotes"` línea que agregamos a nuestra `Quotes#index` vista anteriormente, estamos suscritos al stream y, por lo tanto, recibiremos el HTML que se transmite.

Lo segundo que notamos es que el HTML transmitido está en formato Turbo Stream. Le indica a Turbo que envíe `"prepend"` el contenido del archivo `<template>` al destino `"quotes"` . ¡De hecho, eso es lo que le indicamos al `Quote` modelo que hiciera!

La tercera y última cosa que debemos notar es que el HTML contenido en el `<template>` fue generado por el `quotes/_quote.html.erb` parcial para la cita que se acaba de crear. Cuando Turbo recibe esta plantilla en el frontend, la anexará al nodo DOM con el id de `"quotes"` .

Vamos a esbozar este comportamiento. Nuestra `Quotes#index` página ahora se ve así:



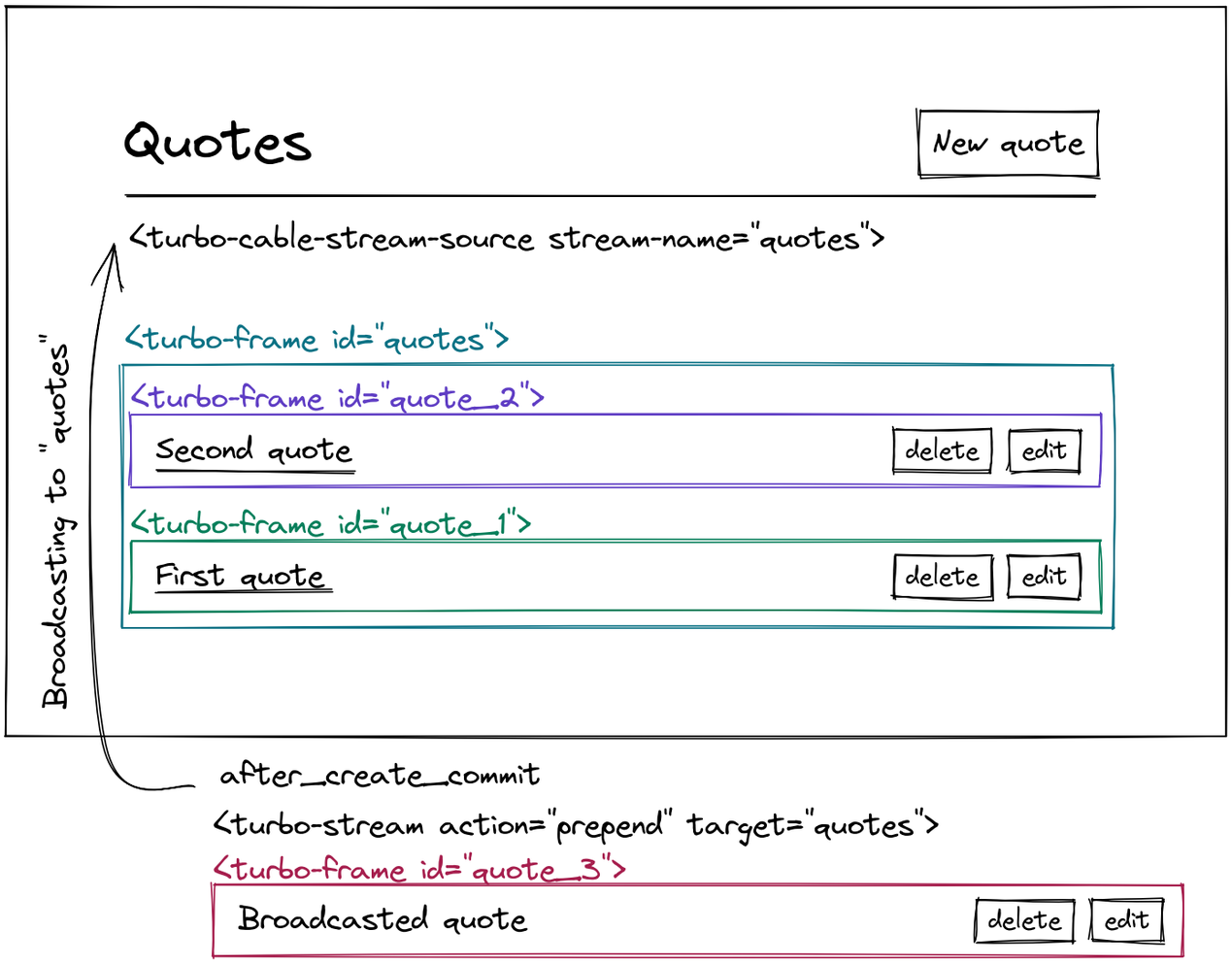
Boceto de la página de citas#índice

Ahora imaginemos que un colega crea una nueva cotización.

Gracias a la `after_create_commit` devolución de llamada, el `broadcasts_prepend_to` método se llama cuando la cotización creada se agrega a la base de datos.

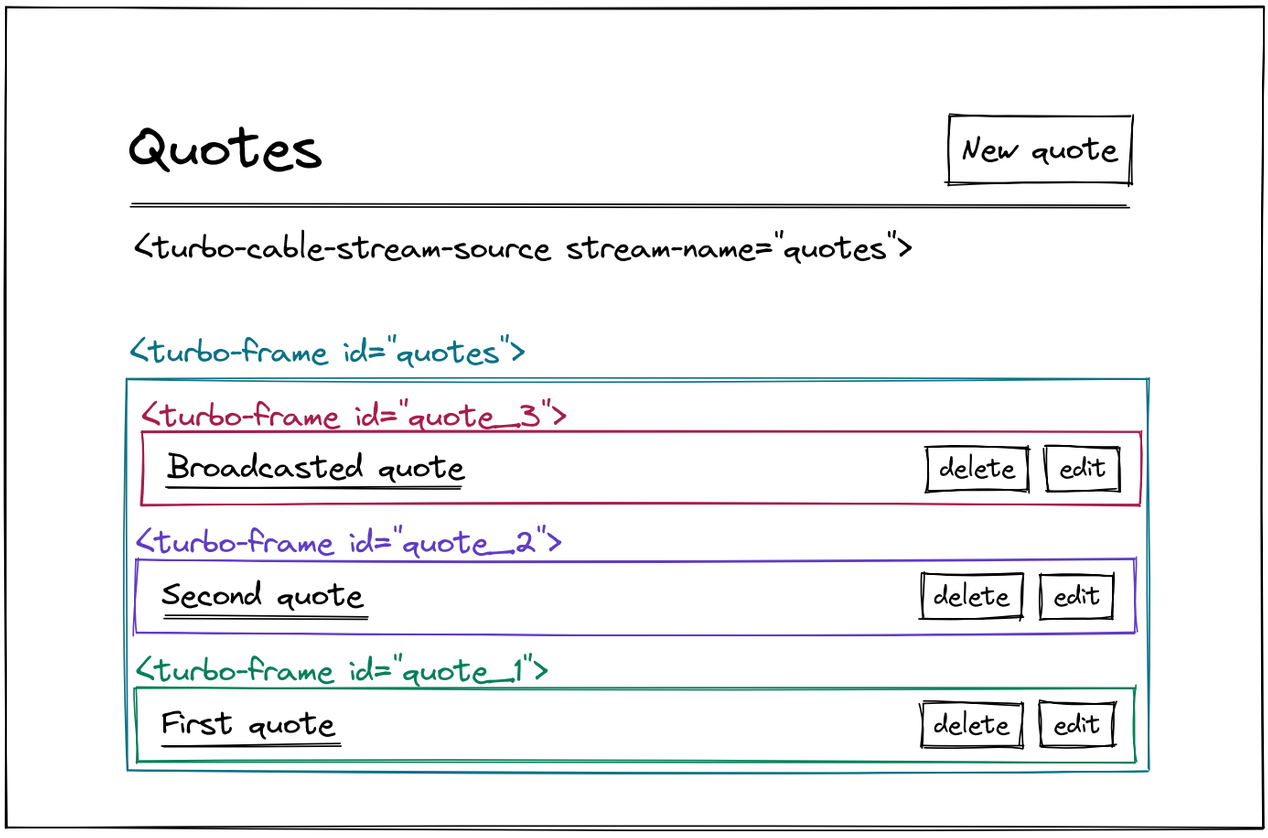
Nos suscribimos a dichas transmisiones en la `Quotes#index` página mediante el `turbo_stream_from` método.

Estas dos líneas de código se describen en el siguiente esquema:



Boceto de la página de índice de citas que recibe la transmisión

En nuestro navegador, deberíamos ver que una cita llamada "Cita transmitida" se ha añadido al principio de la lista de citas en tiempo real:



Boceto de la página Quotes#index con la cita creada antepuesta a la lista

Gracias a Turbo Rails, que está basado en Action Cable, esos cambios se pueden reflejar en tiempo real para todos los usuarios suscritos al canal correcto con el

nombre de transmisión correcto. ¡No tuvimos que actualizar la página para ver el cambio! ¡Con solo dos líneas de código, convertimos nuestra aplicación en una aplicación en tiempo real!

Prueba de Turbo Streams con dos ventanas del navegador

Otra forma de comprobar que todo funciona como se espera es abrir dos ventanas del navegador en la `Quotes#index` página y ponerlas una al lado de la otra. En una de las dos ventanas, vamos a crear una cita. Deberíamos ver que el cambio se refleja inmediatamente en la otra ventana sin tener que actualizar la página.

¡Como podemos ver, nuestra aplicación es reactiva como se esperaba!

Convenciones y azúcar sintáctica de Turbo Streams

Podemos reducir la cantidad de código que escribimos en esta primera parte del capítulo en el `Quote` modelo:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

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

Como podemos ver arriba, especificamos el nombre del objetivo `"quotes"` gracias a la `target: "quotes"` opción. De forma predeterminada, la opción de objetivo será igual a `model_name.plural` , que es igual a `"quotes"` en el contexto de nuestro `Quote` modelo. Gracias a esta convención, podemos eliminar la `target: "quotes"` opción:

```
# app/models/quote.rb

class Quote < ApplicationRecord
```

```
  # All the previous code

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

Hay otras dos convenciones que podemos usar para acortar nuestro código. En esencia, Turbo tiene un valor predeterminado tanto para la opción `partial` como para la `locals` opción.

El `partial` valor predeterminado es igual a llamar `to_partial_path` a una instancia del modelo, que por defecto en Rails para nuestro `Quote` modelo es igual a `"quotes/quote"`.

El `locals` valor predeterminado es igual a `{ model_name.element.to_sym => self }` que, en el contexto de nuestro `Quote` modelo, es igual a `{ quote: self }`.

Estos son precisamente los valores que pasamos como opciones, por lo que el código siguiente es equivalente al que teníamos antes:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

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

Usando las convenciones de Ruby on Rails, ¡hicimos nuestra aplicación en tiempo real con dos líneas de código (muy cortas)!

Ahora que entendemos cómo funciona Turbo Streams, será sencillo finalizar nuestro CRUD en tiempo real en el `Quote` modelo.

Actualizaciones de cotizaciones de transmisión con Turbo Streams

Ahora que nuestra función de creación de cotizaciones en tiempo real está funcionando, agreguemos la misma función para las actualizaciones de

cotizaciones.

Instruyamos a nuestro modelo para que también transmita actualizaciones sobre las cotizaciones:

```
# 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
```

¡Eso es todo! ¡Ya funciona si hacemos pruebas en la consola o con dos ventanas del navegador! Para entender cómo, hagamos pruebas en la consola de Rails:

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

Al igual que antes, al crear una cotización, observamos en los registros de la consola que la cotización se actualiza en la base de datos y que la transacción se confirma:

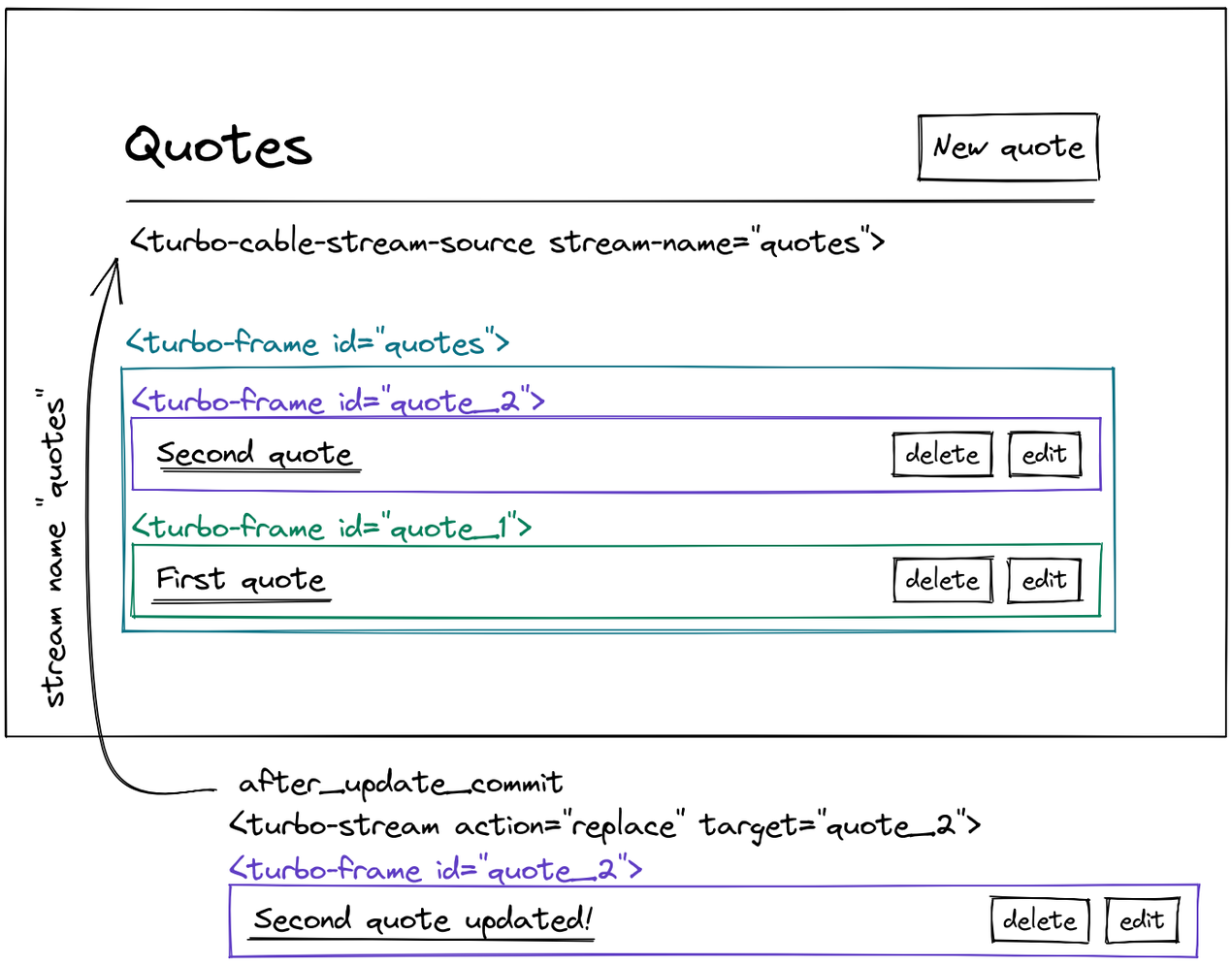
```
Quote Load (0.3ms)  SELECT "quotes".* FROM "quotes" ORDER BY "quotes"."i
TRANSACTION (0.0ms)  begin transaction
Quote Update (0.3ms)  UPDATE "quotes" SET "name" = ?, "updated_at" = ? W
TRANSACTION (1.6ms)  commit transaction
```

Cuando se confirma la transacción, se activa la `after_update_commit` devolución de llamada en el modelo y se llama al método: `Quote broadcast_replace_to`

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

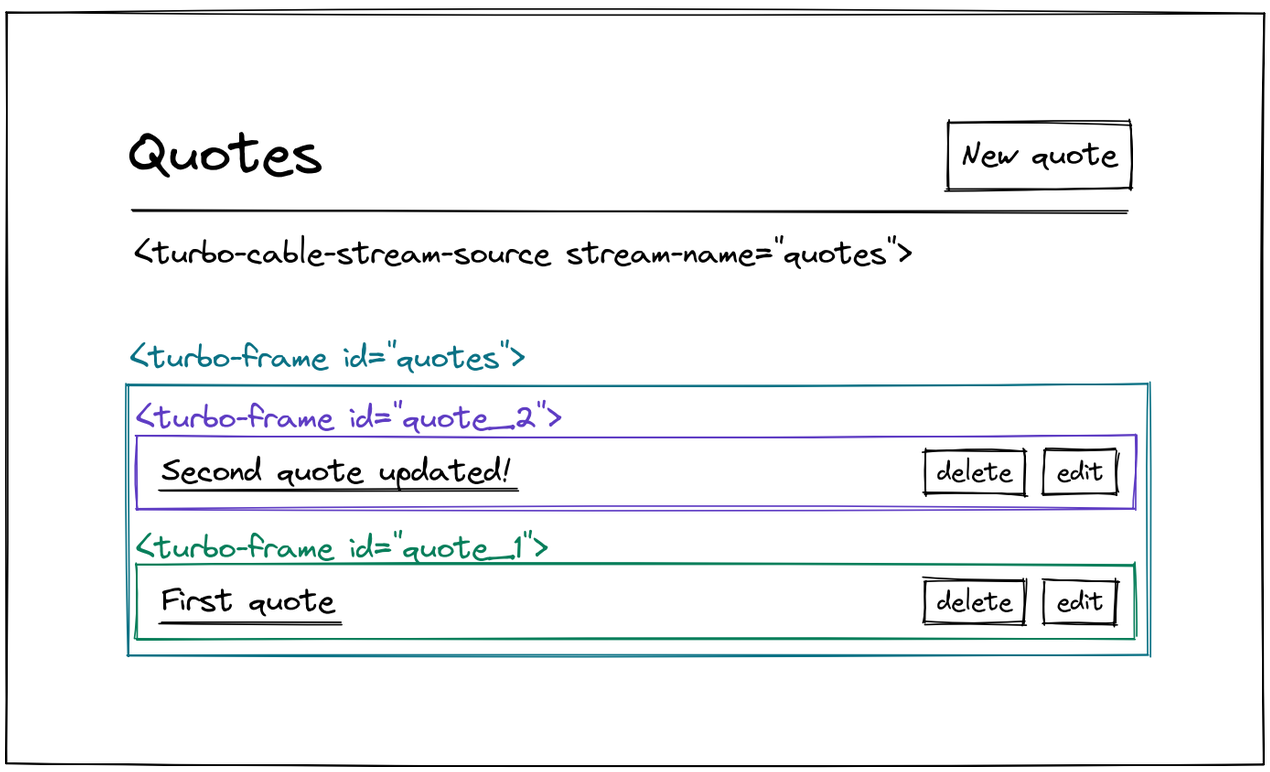
Al igual que la última vez, podemos ver que el HTML del `quotes/quote` parcial se transmite al `"quotes"` flujo. La principal diferencia es que esta vez, la acción es `"replace"` y no `"prepend"` , y que el `target` nodo DOM es la tarjeta de cita

con el id de `"quote_908005754"` donde `"908005754"` es el id de la cita actualizada:



Boceto de la página de índice de citas que recibe la transmisión

Turbo intercepta el HTML recibido y la tarjeta de cotización se reemplaza:



Boceto de la página de índice de citas con la cita actualizada

La última característica que queremos implementar es que nuestra aplicación detecte en tiempo real cuando un usuario elimine una cita. ¡Eso es lo que haremos en la siguiente sección!

Eliminación de citas de transmisión con Turbo Streams

Instruyamos a nuestro `Quote` modelo para que transmita los cambios cuando se elimine una cita de la base de datos. Al igual que antes, esto se hace mediante una devolución de llamada en el modelo:

```
# 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
```

Probemos la función inmediatamente en la consola de Rails para asegurarnos de que funciona como se espera. Asegurémonos de que tenemos una cita en nuestra base de datos local que podemos destruir y ejecutemos el siguiente comando:

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

Como podemos ver en el navegador, ¡funciona como se espera! Analicemos los registros para entender por qué.

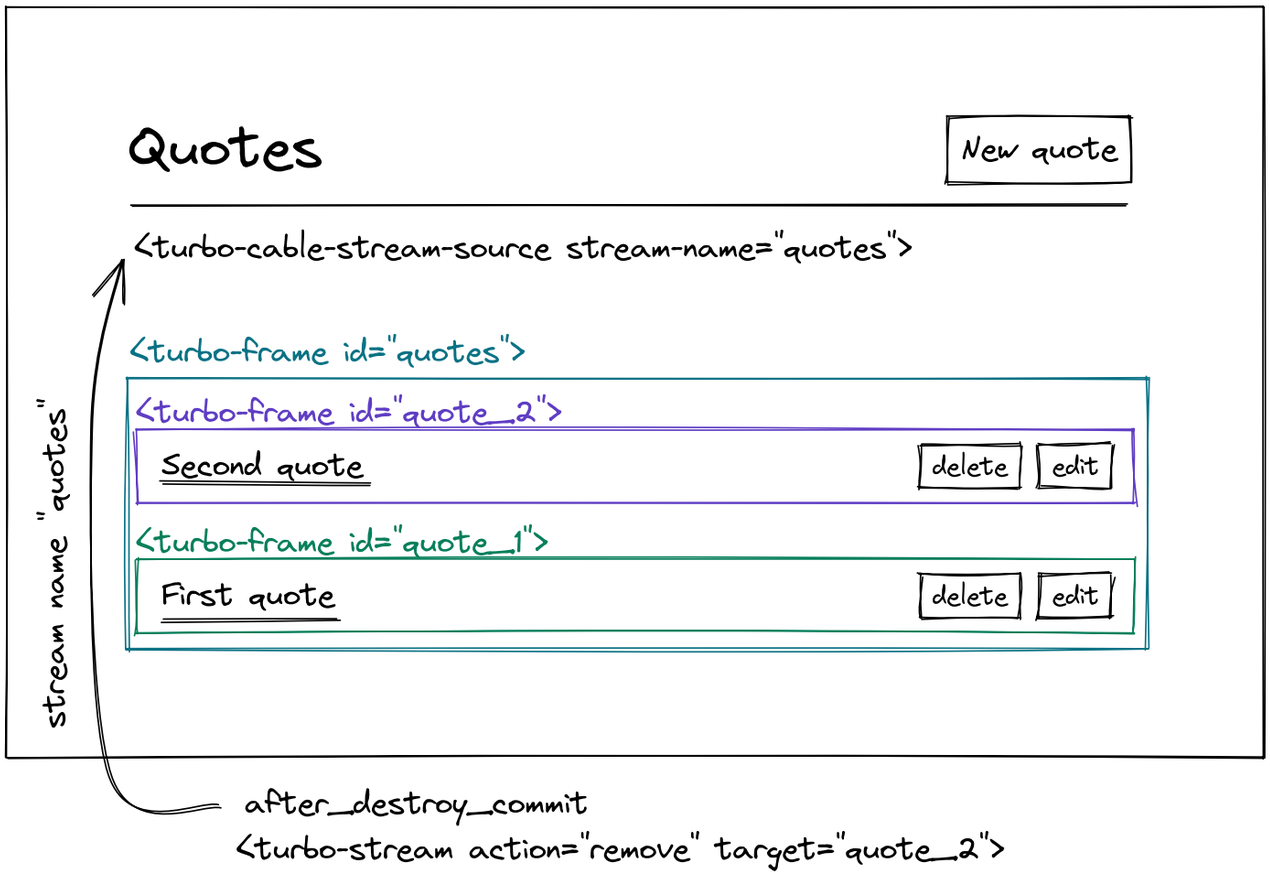
```
Quote Load (0.3ms)  SELECT "quotes".* FROM "quotes" ORDER BY "quotes"."i
TRANSACTION (0.1ms)  begin transaction
Quote Destroy (0.4ms)  DELETE FROM "quotes" WHERE "quotes"."id" = ?  [["
TRANSACTION (1.4ms)  commit transaction
```

En esta primera parte de los logs, podemos ver que la última cotización se recupera de la base de datos y luego se destruye en una transacción. Cuando la transacción termina, `after_destroy_commit` se activa la devolución de

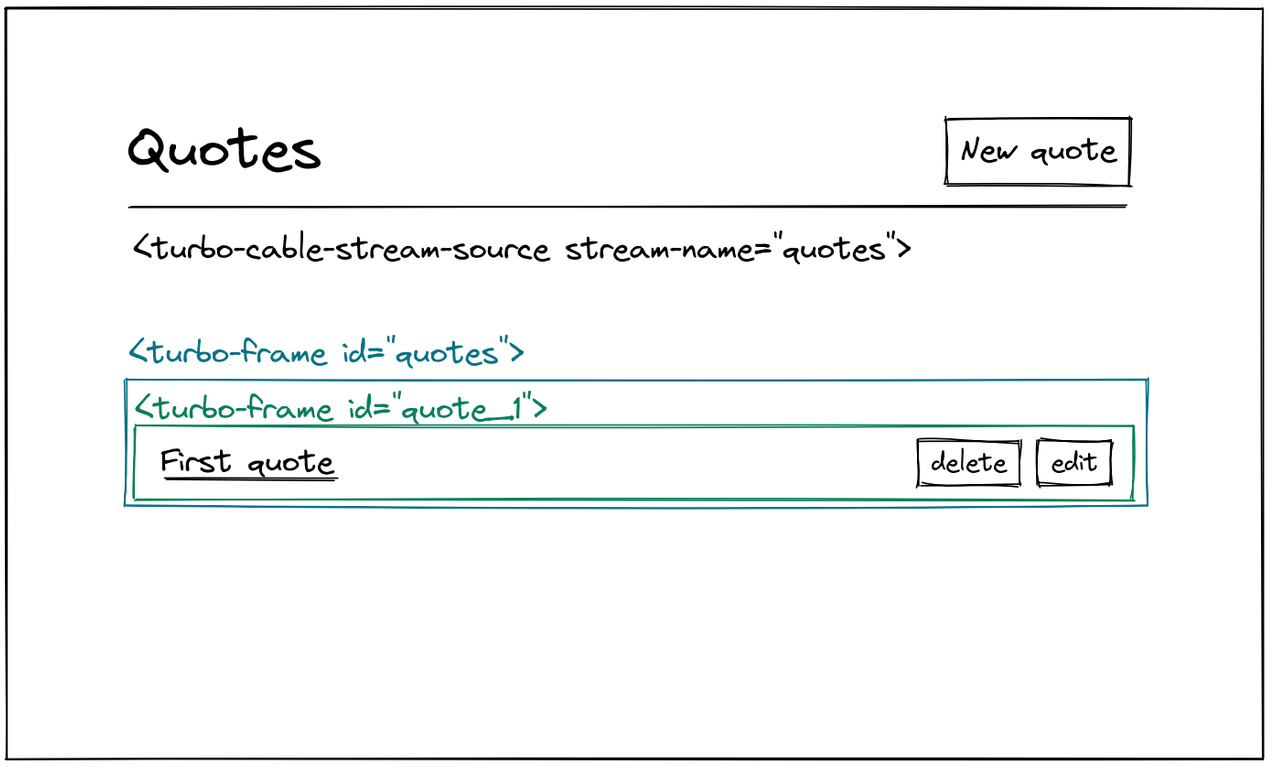
llamada desde nuestro `Quote` modelo y se llama al `broadcast_remove_to` método:

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

El canal transmite algunos HTML `"quotes"` a nuestros usuarios. Esta vez, este HTML solo le indica a Turbo que elimine el nodo DOM de id `"quote_908005754"` , donde `"908005754"` es el id de la base de datos de la cita que se acaba de eliminar:



Como resultado, la cita desaparece de la `Quotes#index` página para todos los usuarios que están en la `Quotes#index` página:



Boceto de la página de índice de citas con la cita eliminada de la lista

Acabamos de finalizar nuestro CRUD en tiempo real sobre el `Quote` modelo. ¡Es emocionante! Antes de terminar este capítulo y pasar al siguiente, debemos hablar sobre el rendimiento.

Cómo hacer que la transmisión sea asincrónica con ActiveJob

Nuestro `Quote` modelo actualmente luce así:

```
# 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
```

Es posible mejorar el rendimiento de este código haciendo que la parte de transmisión sea asíncrona mediante trabajos en segundo plano. Para ello, solo tenemos que actualizar el contenido de nuestros callbacks para utilizar sus equivalentes asíncronos:

```
# 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
```

Nota : El `broadcast_remove_later_to` método no existe porque como la cita se elimina de la base de datos, sería imposible para un trabajo en segundo plano recuperar esta cita en la base de datos más tarde para realizar el trabajo.

Para ver la diferencia, abramos nuestra consola Rails y creemos una nueva cita:

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

Observando los logs detenidamente, veremos que el resultado es el mismo que cuando creamos una cotización, pero la parte de transmisión se realiza de forma asincrónica. Como podemos ver, a `Turbo::Streams::ActionBroadcastJob` se pone en cola con todos los datos necesarios para realizar la transmisión posteriormente.

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

Luego se realiza el trabajo para renderizar el HTML del `quotes/_quote.html.erb` parcial tal como antes:

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

La transmisión de Turbo Streams de forma asincrónica es el método preferido por razones de rendimiento.

Más azúcar sintáctico

Si tuviéramos varios modelos en tiempo real, notaríamos que las devoluciones de llamadas que escribimos en esos modelos son muy similares. Ruby on Rails es un marco fantástico, por lo que hay algo de azúcar sintáctica para evitar escribir esas tres devoluciones de llamadas todo el tiempo. Reemplacémoslas con una versión equivalente y más corta en nuestro `Quote` modelo:

💎 Trenes calientes

```
# 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
```

Las tres devoluciones de llamada son equivalentes a una sola línea de código . Analizaremos por qué necesitamos una lambda en el próximo capítulo sobre Turbo Streams y seguridad. Por ahora, comprendamos que esto significa que queremos transmitir las creaciones, actualizaciones y eliminaciones de citas a la `"quotes"` transmisión de forma asincrónica.

Nuestra `Quote` implementación del modelo final se ve así:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

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

Envolver

Transformar nuestra aplicación en una en tiempo real sólo tomó dos líneas de código, gracias a Turbo Rails.

1. En nuestro `Quote` modelo, configuramos tres devoluciones de llamadas para transmitir creaciones, actualizaciones y eliminaciones en la `"quotes"` transmisión. Gracias al `broadcasts_to` método, esas tres devoluciones de llamadas se pueden definir en una línea.
2. En nuestra `Quotes#index` opinión, mencionamos explícitamente que queremos suscribirnos a los cambios que se transmiten en el `"quotes"` stream.

Turbo se encarga del resto, haciendo que sea un placer trabajar con nuestra aplicación Ruby on Rails.

En el próximo capítulo, abordaremos Turbo Streams y la seguridad. Analizaremos cómo funciona la seguridad con Turbo Streams para garantizar que no transmitamos datos privados a los usuarios incorrectos.

← anterior Siguiente →