Hotrails

← Back to the list of chapters

Turbo Frames and Turbo Stream templates

Published on January 28, 2022

In this chapter, we will learn how to slice our page into independent parts thanks to Turbo Frames and the Turbo Stream format. After reading this chapter, all the CRUD actions on quotes will happen on the quotes index page.

What we will build in this chapter

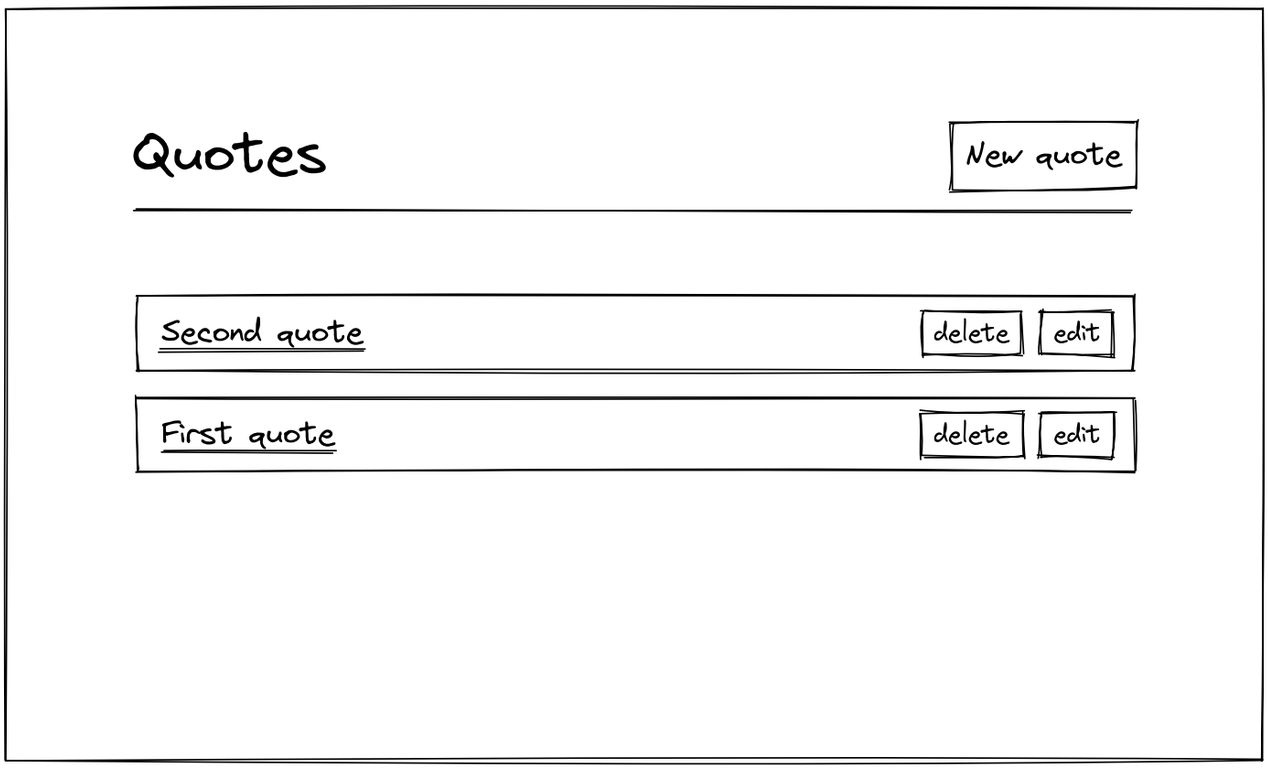
The `#new` and `#edit` actions happen on different pages in our current quote editor. When we are on the `Quotes#index` page:

- Clicking on the "New quote" button opens a completely different page containing only a title and a form to create a new quote.
- Clicking on the "Edit" button for a quote opens another page containing only a title and a form to edit the quote.

We would like to avoid this *context switch*. Instead, we would like to perform those two actions directly on the `Quotes#index` page, just like in the final quote editor. That's what we are going to learn to do in this chapter. It will require only a few lines of code thanks to the incredible power of Turbo Frames and Turbo Streams.

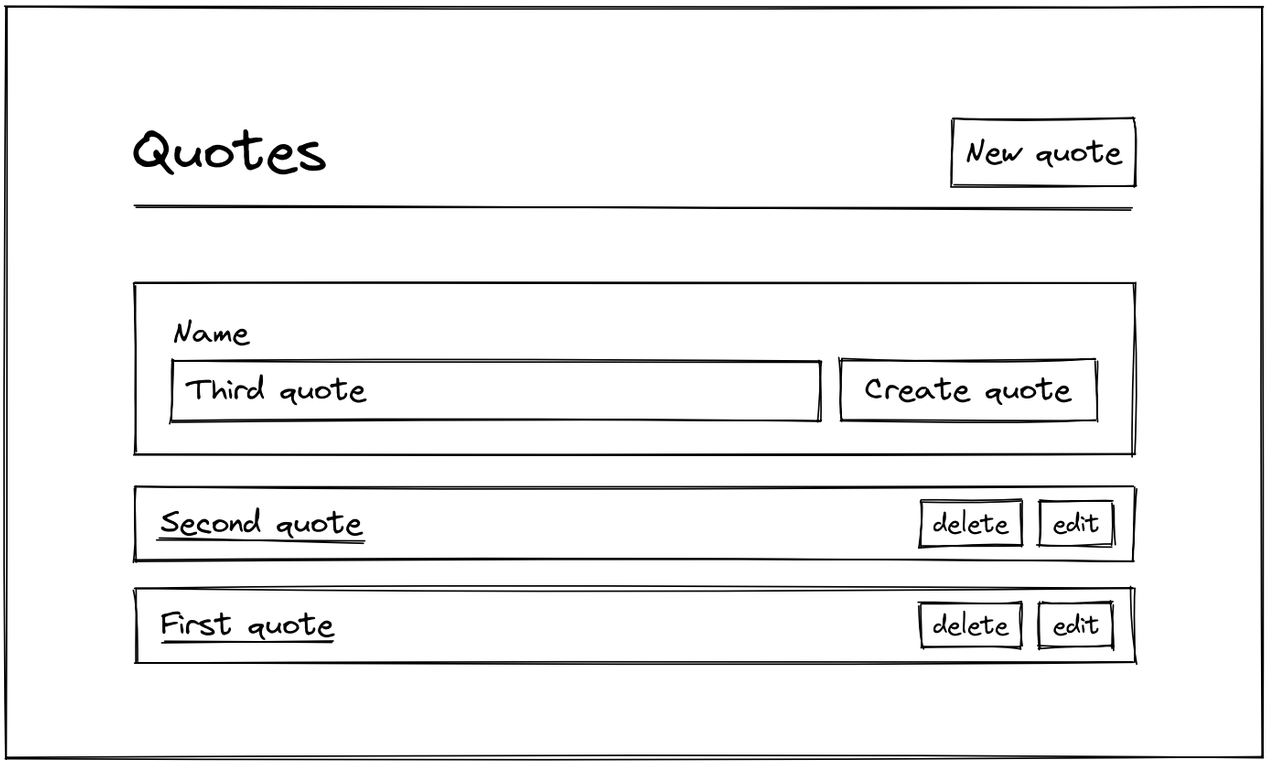
Before practicing our Turbo Frame skills, let's make a few sketches of what we will build and update our system tests.

Our `Quotes#index` page currently looks like this:



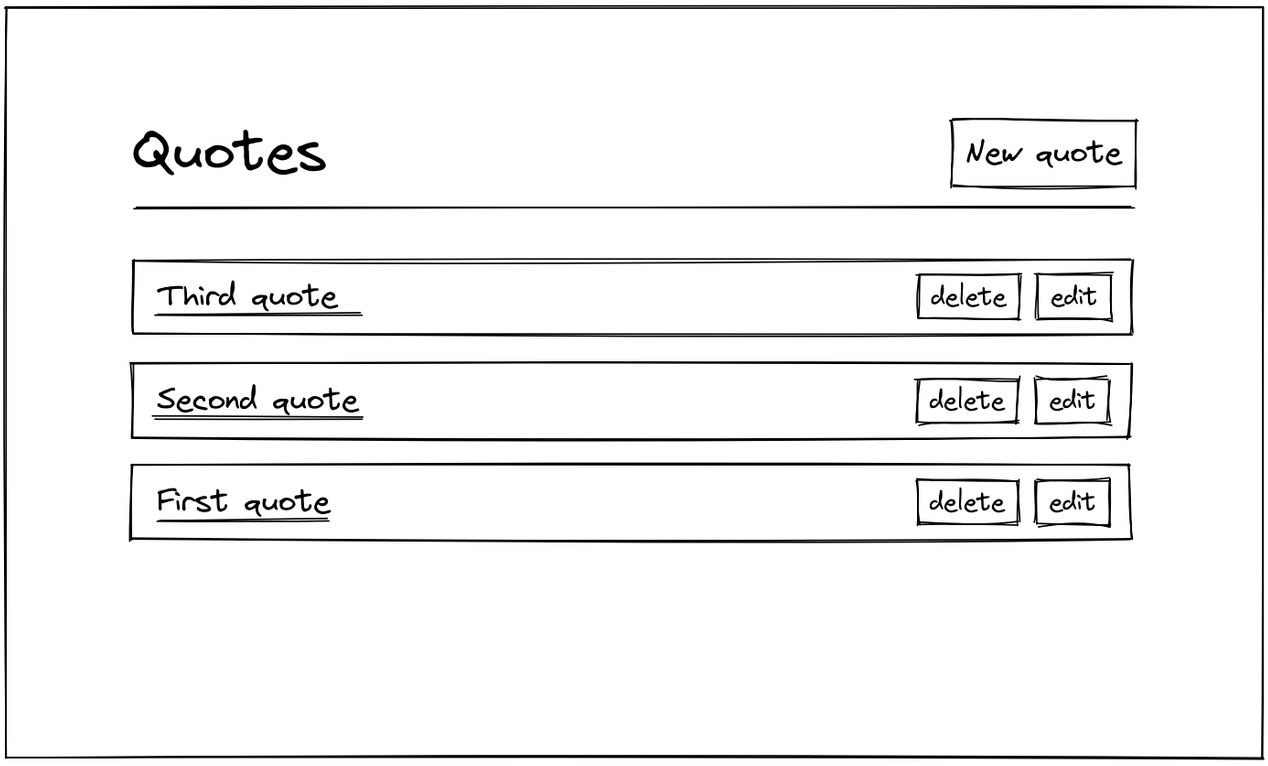
Sketch of the Quotes#index page

When clicking on the "New quote" button, we want the new quote form to be appended to the page right below the header:



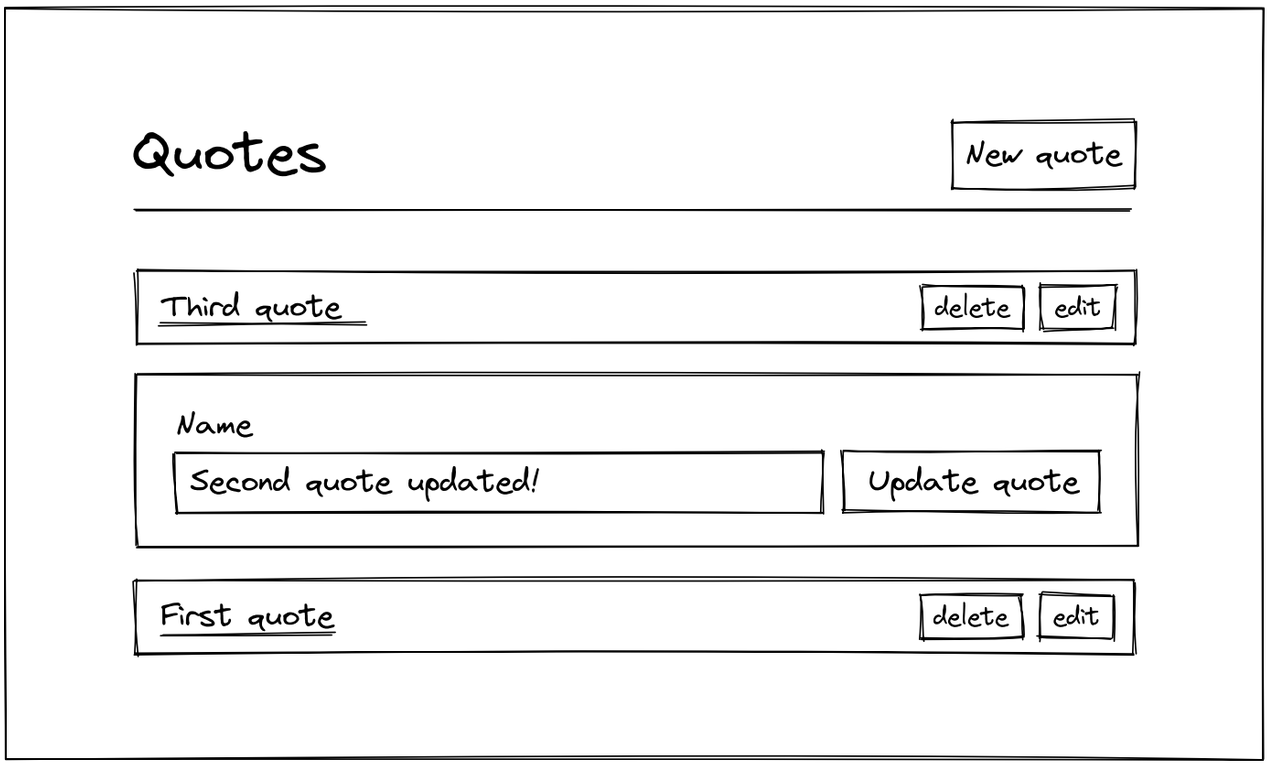
Sketch of the Quotes#index page with the new quote form

When clicking on the "Create quote" button, the created quote should be prepended to the quotes list:



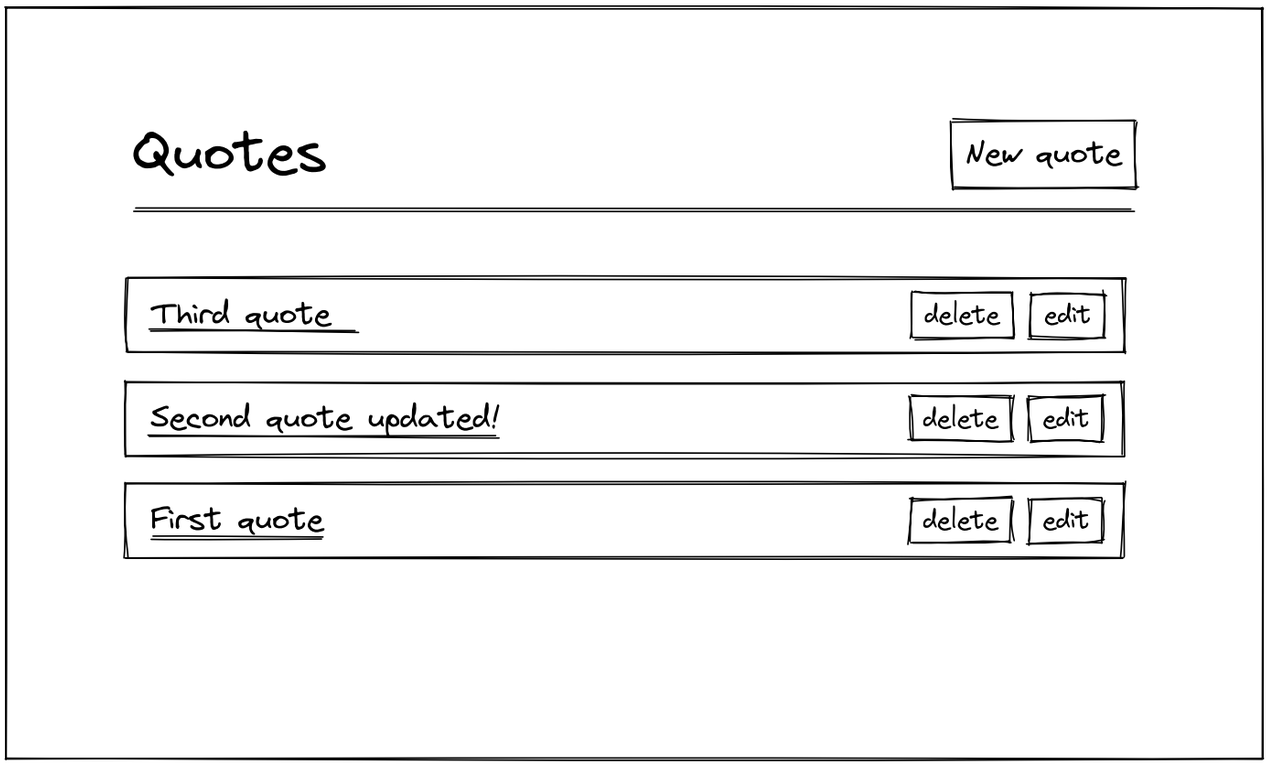
Sketch of the Quotes#index page with the created quote prepended to the list

When clicking on the "Edit" button **for the second quote**, the quote card should be replaced by a form to edit the corresponding quote:



Sketch of the Quotes#index page with a form to edit the second quote

When clicking on the "Update quote" button, the form for the second quote should be replaced by the updated quote:



Sketch of the Quotes#index page with the updated quote

The rest of the behavior should remain unchanged:

- Clicking on the "Delete" button for a quote should remove it from the quotes list.
- Clicking on the name of a quote should take us to the `Quotes#show` page.

Let's now update our Capybara system tests to match the desired behavior:

```
# test/system/quotes_test.rb

require "application_system_test_case"

class QuotesTest < ApplicationSystemTestCase
  setup do
    @quote = quotes(:first)
  end

  test "Showing a quote" do
    visit quotes_path
    click_link @quote.name

    assert_selector "h1", text: @quote.name
  end

  test "Creating a new quote" do
    visit quotes_path
    assert_selector "h1", text: "Quotes"
```

```
    click_on "New quote"
    fill_in "Name", with: "Capybara quote"

    assert_selector "h1", text: "Quotes"
    click_on "Create quote"

    assert_selector "h1", text: "Quotes"
    assert_text "Capybara quote"
  end

  test "Updating a quote" do
    visit quotes_path
    assert_selector "h1", text: "Quotes"

    click_on "Edit", match: :first
    fill_in "Name", with: "Updated quote"

    assert_selector "h1", text: "Quotes"
    click_on "Update quote"

    assert_selector "h1", text: "Quotes"
    assert_text "Updated quote"
  end

  test "Destroying a quote" do
    visit quotes_path
    assert_text @quote.name

    click_on "Delete", match: :first
    assert_no_text @quote.name
  end
end
```

If we run the tests now, the two tests corresponding to the creation and the edition of quotes will fail. Our goal is to make them green again with Turbo Frames and Turbo Streams. Ready to learn how they work? Let's dive in!

What are Turbo Frames?

Turbo Frames are independent pieces of a web page that can be appended, prepended, replaced, or removed without a complete page refresh and writing a

single line of JavaScript!

In this section, we will learn everything there is to know about Turbo Frames with a series of small examples. Then we will come back to our quotes and implement the desired behavior with only a few lines of code.

Let's create our first Turbo Frame. To create Turbo Frames, we use the `turbo_frame_tag` helper. Let's wrap the header on the `Quotes#index` page in a Turbo Frame with an id of "first_turbo_frame":

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

<main class="container">
  <%= turbo_frame_tag "first_turbo_frame" do %>
    <div class="header">
      <h1>Quotes</h1>
      <%= link_to "New quote", new_quote_path, class: "btn btn--primary"
    </div>
  <% end %>

  <%= render @quotes %>
</main>
```

If we have a look a the DOM, the generated HTML for the Turbo Frame looks like this:

```
<turbo-frame id="first_turbo_frame">
  <div class="header">
    <h1>Quotes</h1>
    <a class="btn btn--primary" href="/quotes/new">New quote</a>
  </div>
</turbo-frame>
```

As we can see, the `turbo_frame_tag` helper creates a `<turbo-frame>` custom element that contains the HTML generated by the content of the block. This custom element has a unique id corresponding to the first argument we passed to the `turbo_frame_tag` helper.

This `<turbo-frame>` HTML tag does not exist in the HTML language. It is a custom element that was created in the Turbo JavaScript library. It intercepts

form submissions and clicks on links within the frame, making those frames independent pieces of your web page!

Now let's click on the "New quote" button and... The frame disappears from the page, and there is an error in the console: *Response has no matching `<turbo-frame id="first_turbo_frame">` element.* Let's explain this strange behavior.

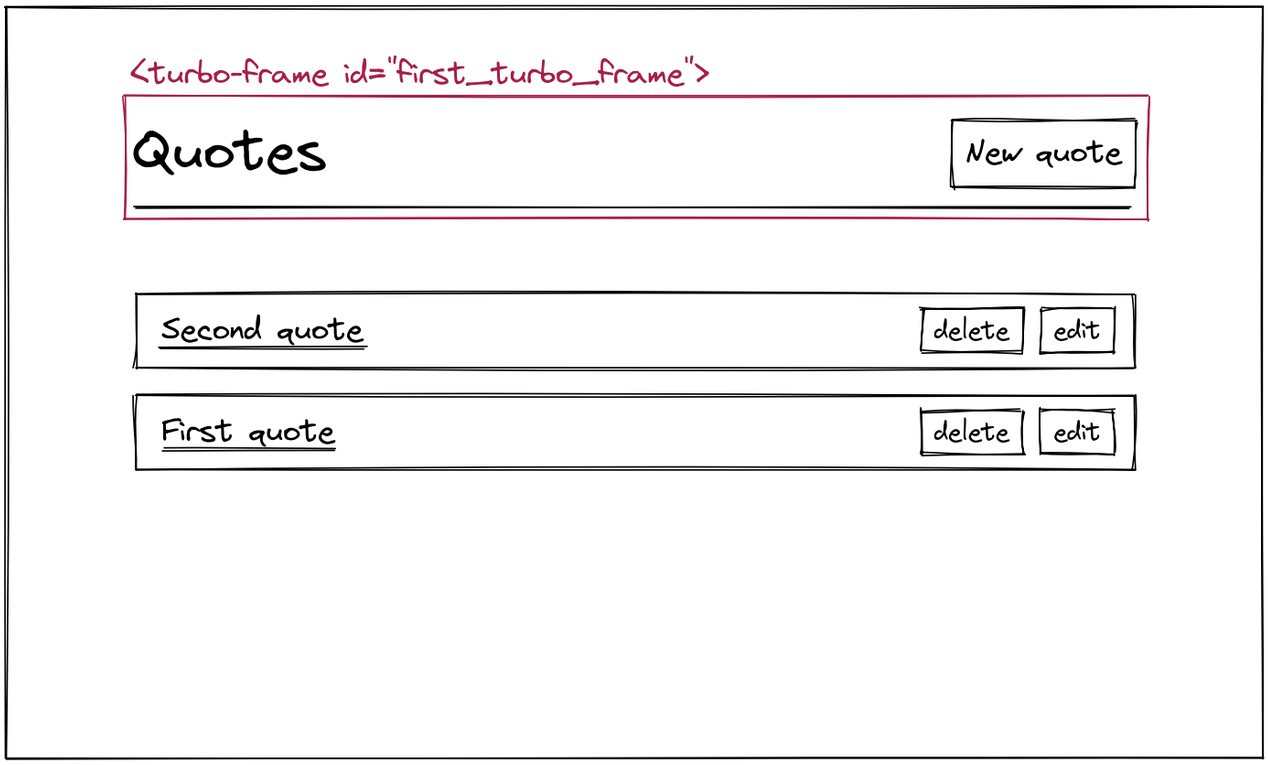
Turbo Frames cheat sheet

In this section, we will explain the rules that apply to Turbo Frames.

Even if the examples are written with links, those rules apply for both links and forms!

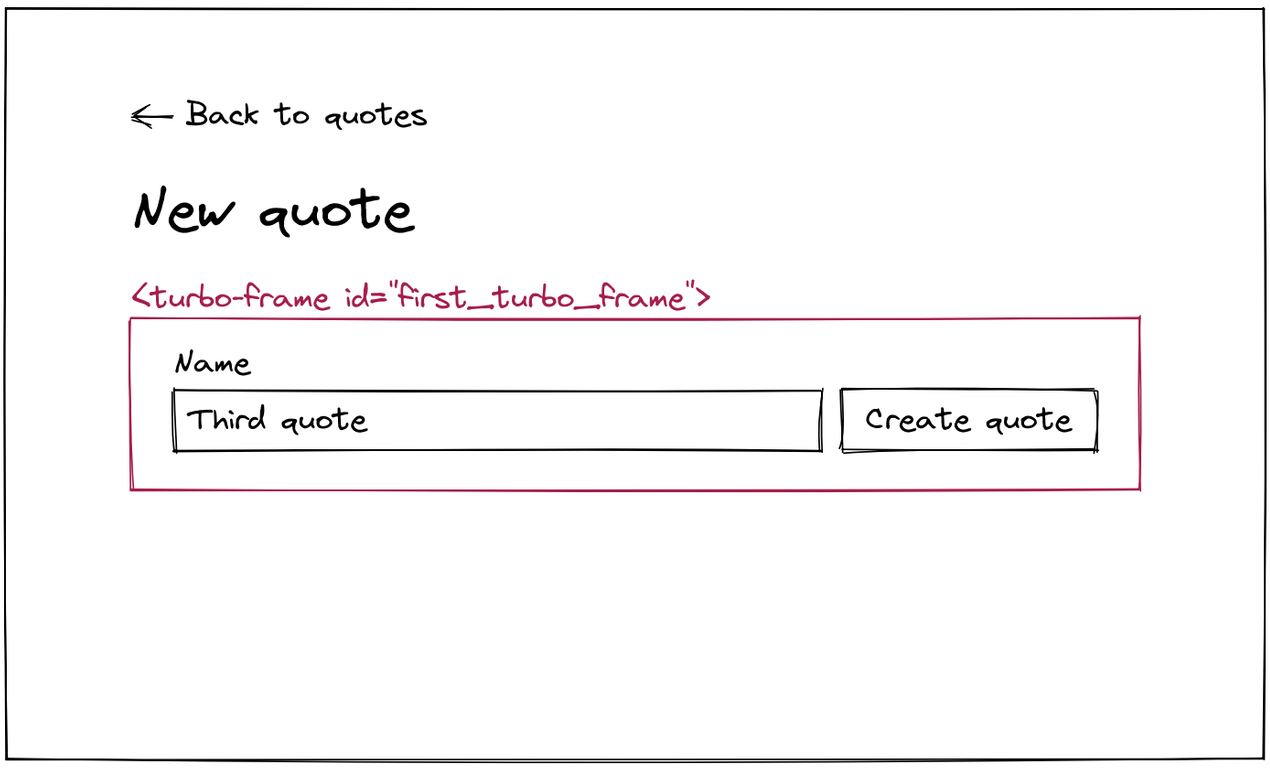
Rule 1: When clicking on a link within a Turbo Frame, Turbo expects a frame of the same id on the target page. It will then replace the Frame's content on the source page with the Frame's content on the target page.

If that's not very clear right now, let's draw a few sketches of the experiment we will carry. Our `Quotes#index` page currently looks like this:



Sketch of the Quotes#index page with our first Turbo Frame

Now, let's wrap a piece of the `Quotes#new` page in a Turbo Frame of the same id. In our example, we will wrap the form in that Turbo Frame like this:



Sketch of the Quotes#new page with our first Turbo Frame

To match the sketches we just draw, let's add the Turbo Frame with the same id on the `Quotes#new` page:

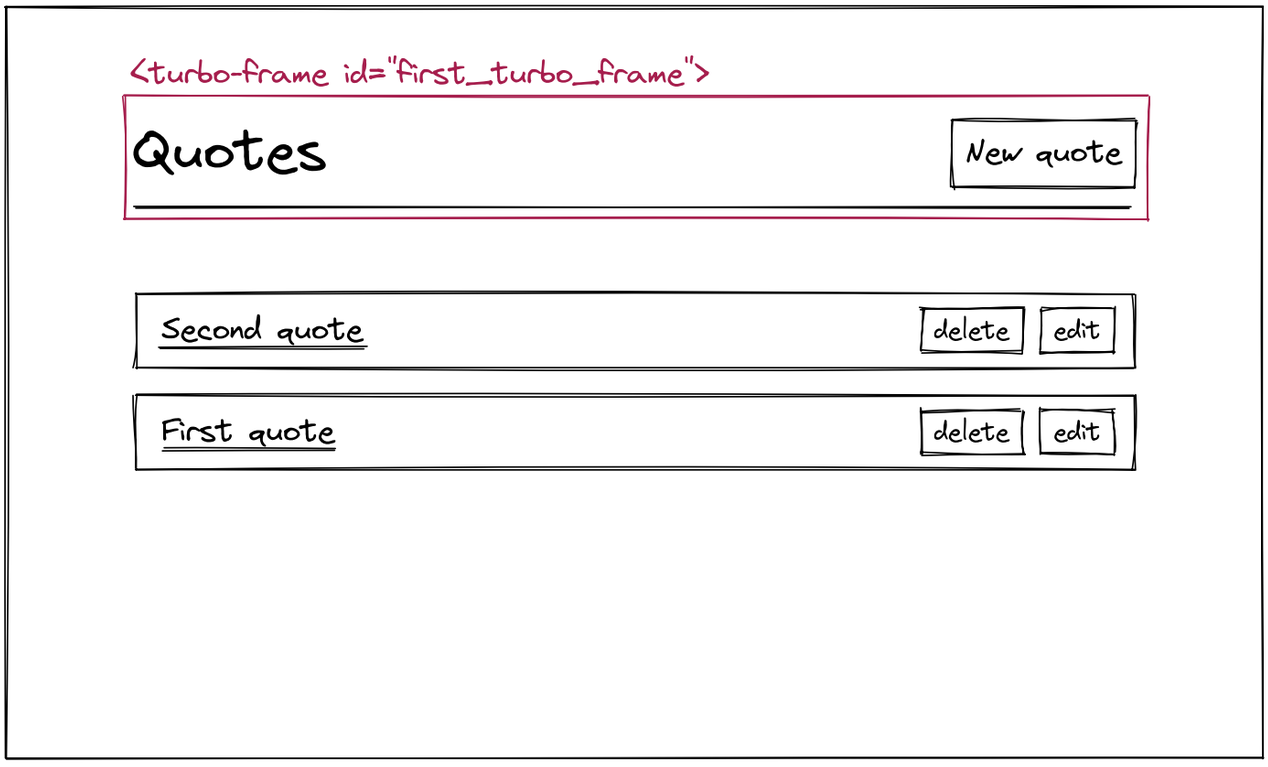
```
<%# app/views/quotes/new.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>

  <div class="header">
    <h1>New quote</h1>
  </div>

  <%= turbo_frame_tag "first_turbo_frame" do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

Let's now experiment in the browser. Let's refresh the `Quotes#index` page and click on the "New quote" button. We can see the content of our Turbo Frame with id `"first_turbo_frame"` on the `Quotes#index` page is replaced by the content of the Turbo Frame on the `Quotes#new` page!

When clicking on a link within a Turbo Frame, if there is a frame with the same id on the target page, Turbo will replace the content of the Turbo Frame of the source page with the content of the Turbo Frame of the target page.

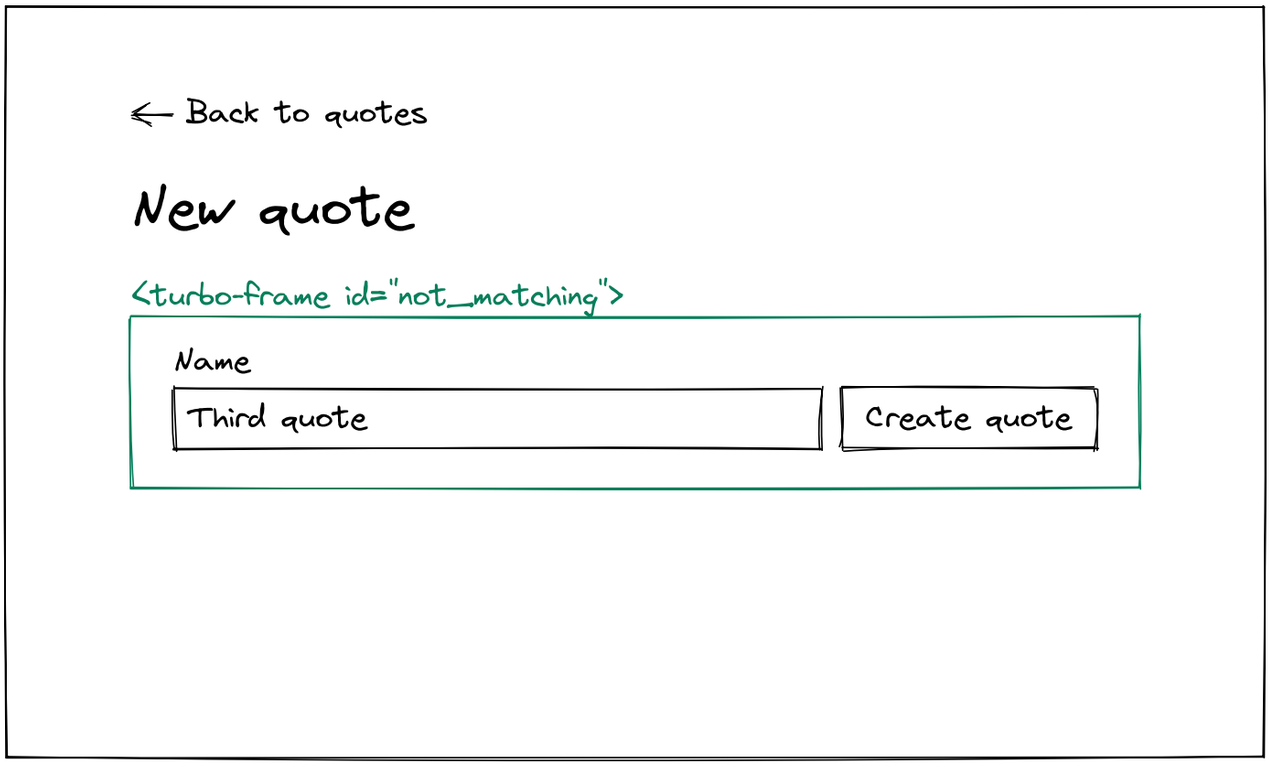
Rule 2: When clicking on a link within a Turbo Frame, if there is no Turbo Frame with the same id on the target page, the frame disappears, and the error *Response has no matching `<turbo-frame id="name_of_the_frame">` element* is logged in the console.

Remember the strange behavior we had when we had no Turbo Frame with the same id on the `Quotes#new` page. That's exactly what this second rule is all about. Our current `Quotes#index` page looks like this:



Sketch of the Quotes#index page with our first Turbo Frame

On the `Quotes#new` page, let's change the id of the Turbo Frame around the form to "not_matching" as described in the sketch below:



Sketch of the Quotes#new page with no matching Turbo Frame

Let's update the markup of the `Quotes#new` page to match our sketches:

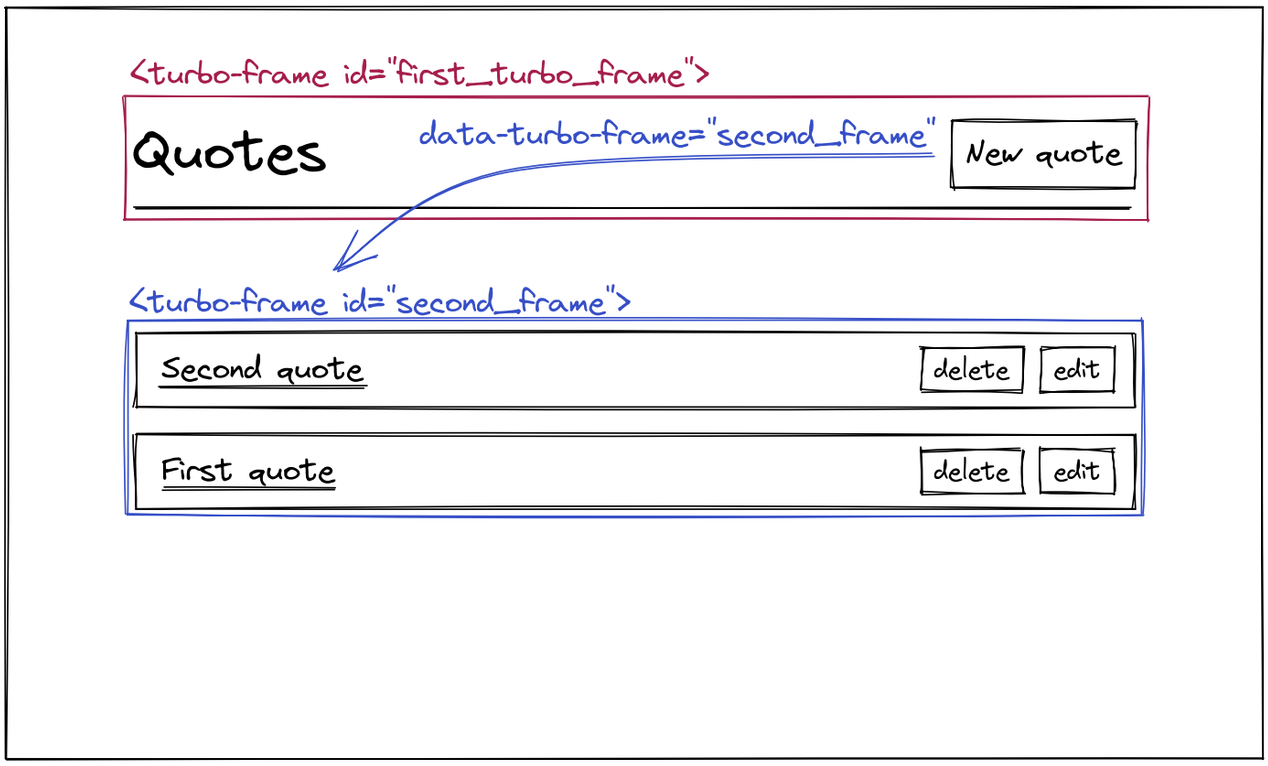
```
<%# app/views/quotes/new.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>

  <div class="header">
    <h1>New quote</h1>
  </div>

  <%= turbo_frame_tag "not_matching" do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

Let's perform the experiment again, navigate to the `Quotes#index` page, refresh the page and click on the "New quote" button. The Turbo Frame with the header disappears, and the error *Response has no matching `<turbo-frame id="name_of_the_frame">` element* is logged in the console as expected.

When clicking on a link within a Turbo Frame, if there is no Turbo Frame with the same id on the target page, Turbo will remove the content of the Turbo Frame from the source page and log an error.

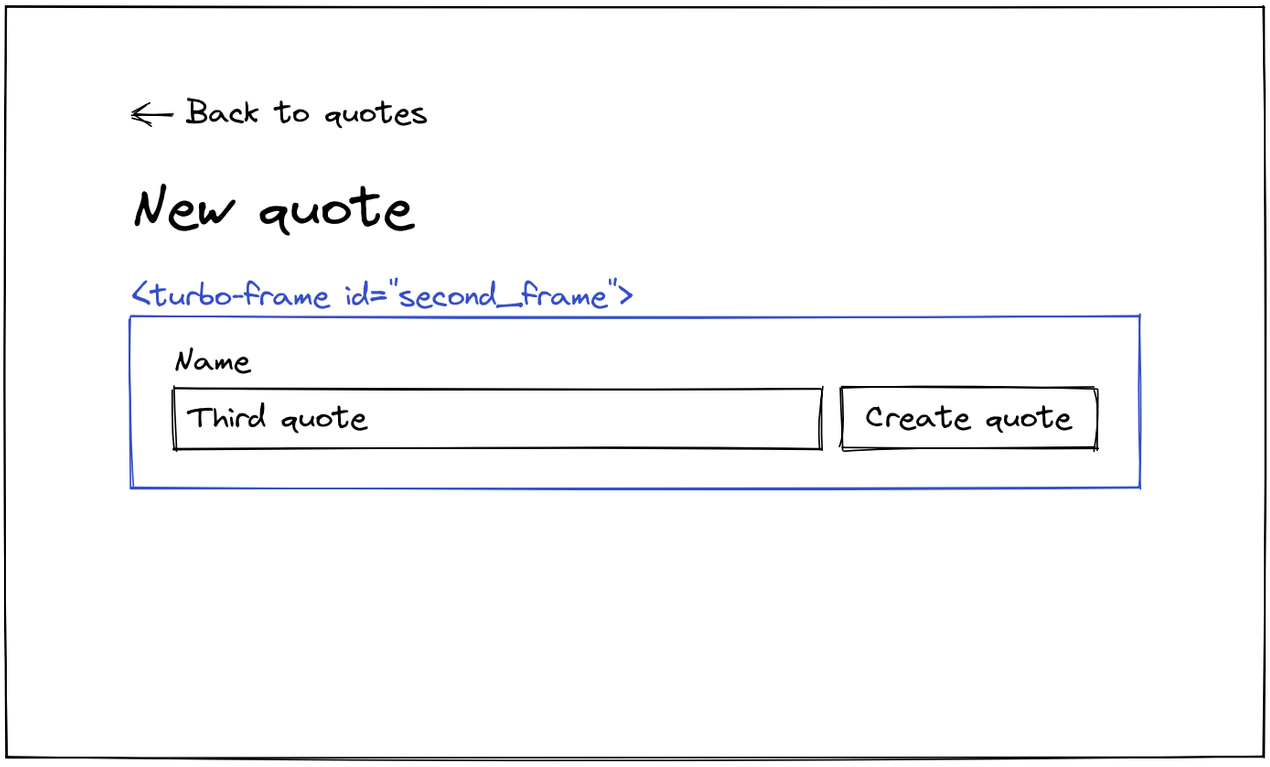
Rules 3: A link can target another frame than the one it is directly nested in thanks to the `data-turbo-frame` data attribute.

This rule is very useful, but we will need more sketches to understand it clearly. Let's first add another Turbo Frame with the id of "second_frame" around the list of quotes in the `Quotes#index` page:



Sketch of the Quotes#index page with our first Turbo Frame and a second Turbo Frame

On the `Quotes#new` page, let's wrap the form in a frame with the same id as the second frame:



Sketch of the Quotes#new with the second Turbo Frame

Let's now update our code to match the sketches. On the `Quotes#index` page, we need to add the second Turbo Frame and the `data-turbo-frame` data attribute with the same id as this second Turbo Frame:

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

<main class="container">
  <%= turbo_frame_tag "first_turbo_frame" do %>
    <div class="header">
      <h1>Quotes</h1>
      <%= link_to "New quote",
                  new_quote_path,
                  data: { turbo_frame: "second_frame" },
                  class: "btn btn--primary" %>
    </div>
  <% end %>

  <%= turbo_frame_tag "second_frame" do %>
    <%= render @quotes %>
  <% end %>
</main>
```

On the `Quote#new` page, let's wrap our form in a Turbo Frame of the same name as the second frame:

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

<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>

  <div class="header">
    <h1>New quote</h1>
  </div>

  <%= turbo_frame_tag "second_frame" do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

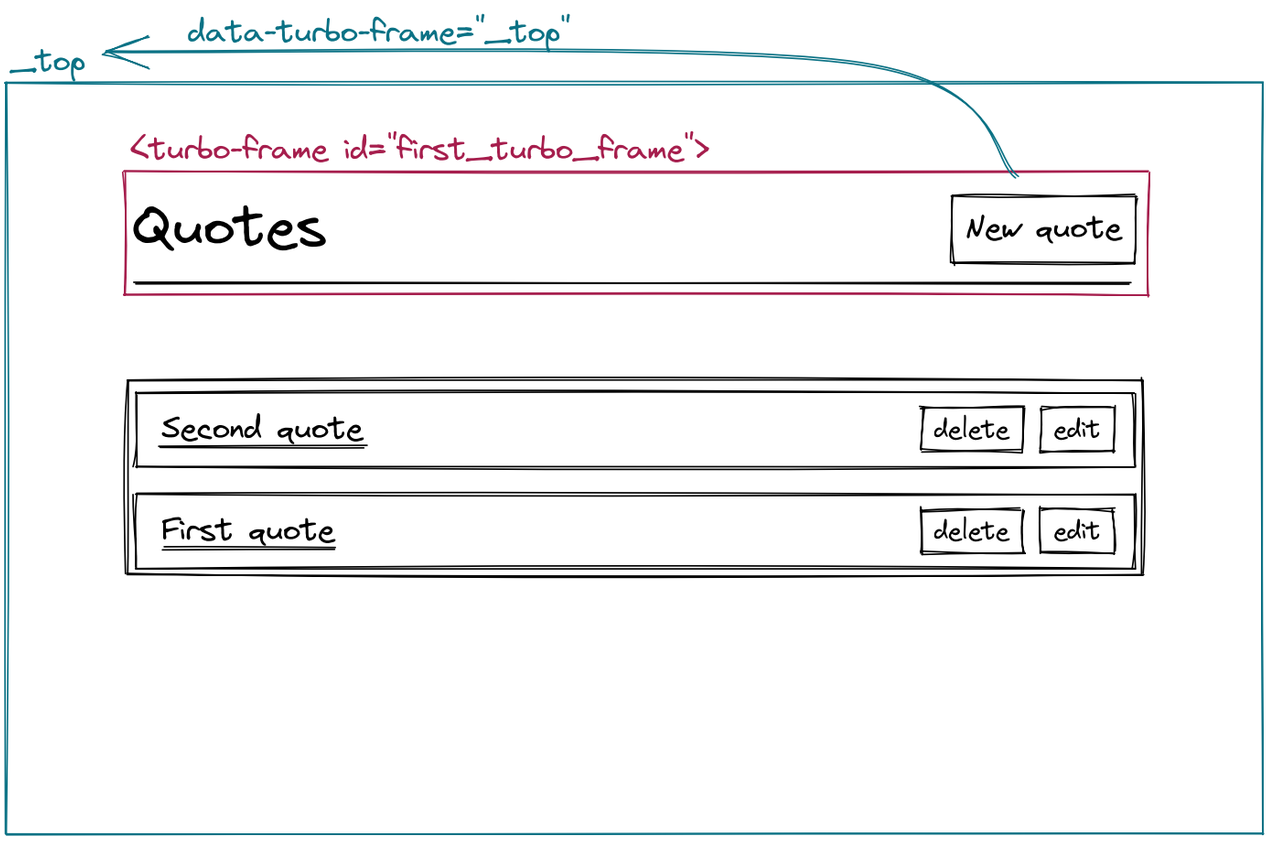
Now let's experiment again. Let's visit the `Quotes#index` page, refresh it, and click on the "New quote" button. We should see our quotes list replaced by the new quote form. This is because our link now targets the second frame thanks to the `data-turbo-frame` attribute.

A link can target a Turbo Frame it is not directly nested in, thanks to the `data-turbo-frame` data attribute. In that case, the Turbo Frame with the same id as the `data-turbo-frame` data attribute on the source page will be replaced by the Turbo Frame of the same id as the `data-turbo-frame` data attribute on the target page.

Note:

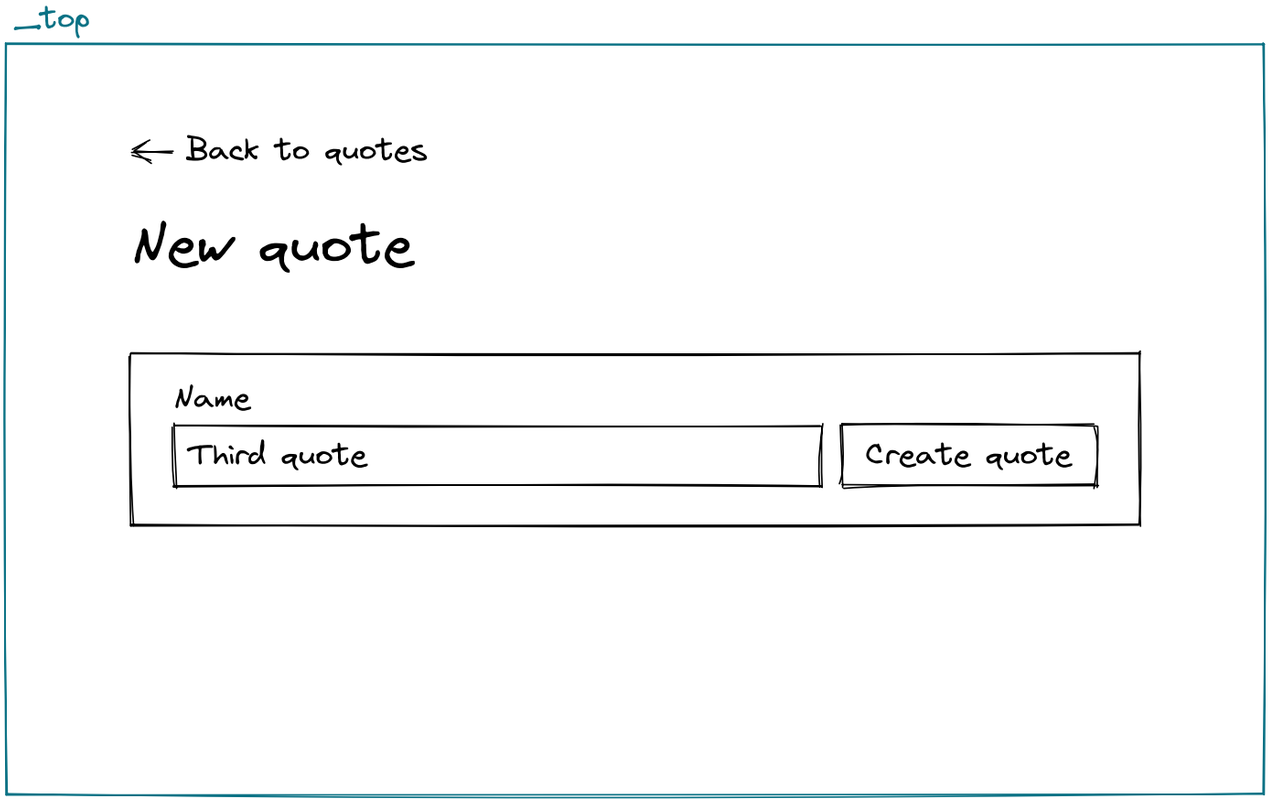
There is a special *frame* called `_top` that represents the whole page. **It's not really a Turbo Frame, but it behaves almost like one, so we will make this approximation for our mental model.**

For example, if we wanted our "New quote" button to replace the whole page, we could use `data-turbo-frame="_top"` like this:



Sketch of the Quotes#index with the _top frame represented

Of course, every page has the "_top" *frame* by default, so our `Quotes#new` page also has it:



Sketch of the Quotes#new page with the _top frame represented

To make our markup match our sketches on the `Quotes#index` page, let's tell our "New quote" link to target the "_top" *frame*:

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

<main class="container">
  <%= turbo_frame_tag "first_turbo_frame" do %>
    <div class="header">
      <h1>Quotes</h1>
      <%= link_to "New quote",
                  new_quote_path,
                  data: { turbo_frame: "_top" },
                  class: "btn btn--primary" %>
    </div>
  <% end %>

  <%= render @quotes %>
</main>
```

We can add whatever we want on the `Quotes#new` page. It does not matter as the browser will replace the whole page. For our example, we will simply go back to our initial state:

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

```
<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>

  <div class="header">
    <h1>New quote</h1>
  </div>

  <%= render "form", quote: @quote %>
</main>
```

Now let's experiment again. Let's navigate to the `Quotes#index` page and click on the "New quote" button. We can see that the whole page is replaced by the content of the `Quotes#new` page.

When using the "_top" keyword, the URL of the page changes to the URL of the target page, which is another difference from when using a regular Turbo Frame.

As we can notice, Turbo Frames are a significant addition to our toolbox as Ruby on Rails developers. They enable us to slice up pages in independent contexts without writing any custom JavaScript.

With those three rules, we have more than enough Turbo Frames knowledge to build our quote editor, but we still need to learn two things:

- How to use Turbo Frames in combination with the `TURBO_STREAM` format
- How to name our Turbo Frames with some good conventions

Let's practice and make our system tests pass! But just before, let's reset our `Quotes#index` page markup to its initial state:

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

<main class="container">
  <div class="header">
    <h1>Quotes</h1>
    <%= link_to "New quote", new_quote_path, class: "btn btn--primary" %
  </div>
```

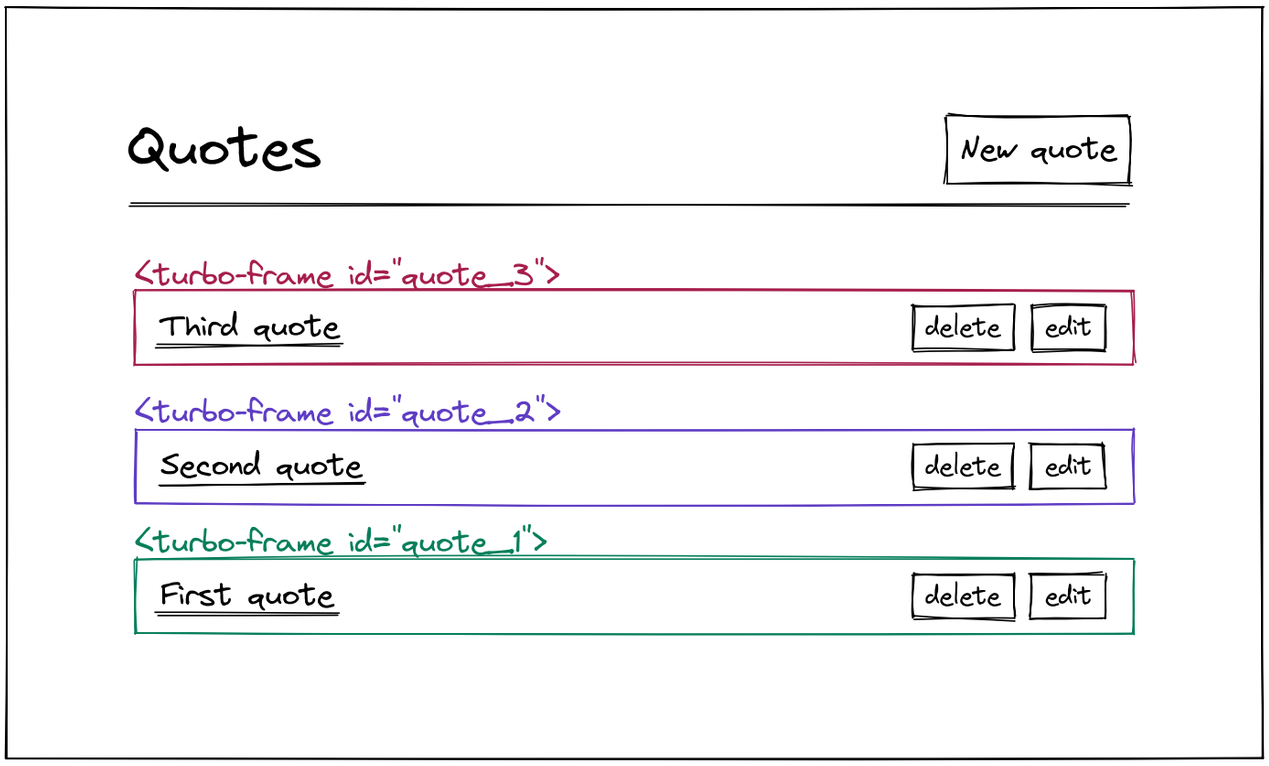
```
  <%= render @quotes %>
</main>
```

Editing quotes with Turbo Frames

Let's start with the quote edition feature, as it is the easiest one of the two.

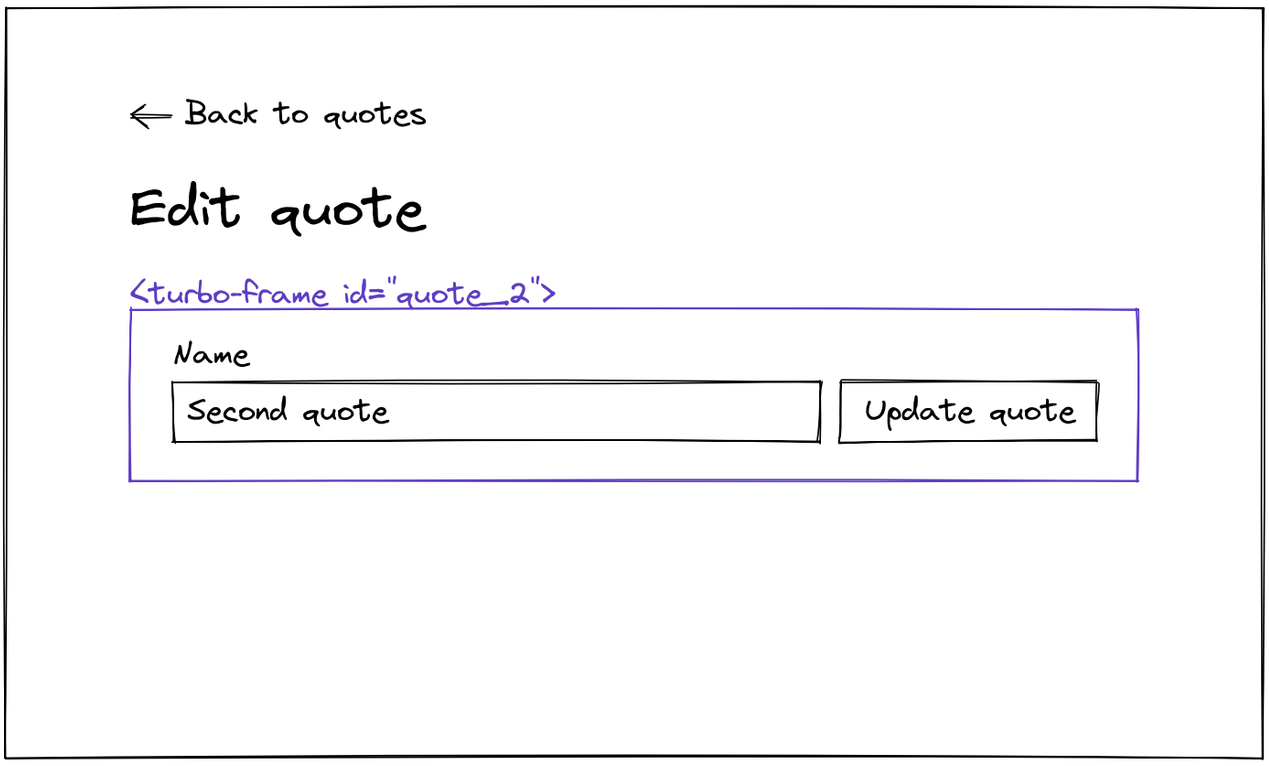
The first thing we need to achieve is that when clicking on the "Edit" button of a quote on the `Quotes#index` page, the card containing the quote will be replaced by a card containing the edition form. Replacing pieces of a web page is precisely the kind of job Turbo Frames can do for us! But what id should we give our Turbo Frames?

On the `Quotes#index` page, each Turbo Frame around each quote card should have a unique id. A good convention is to use the singular name of the model followed by the id of the quote. Let's sketch what it should look like:



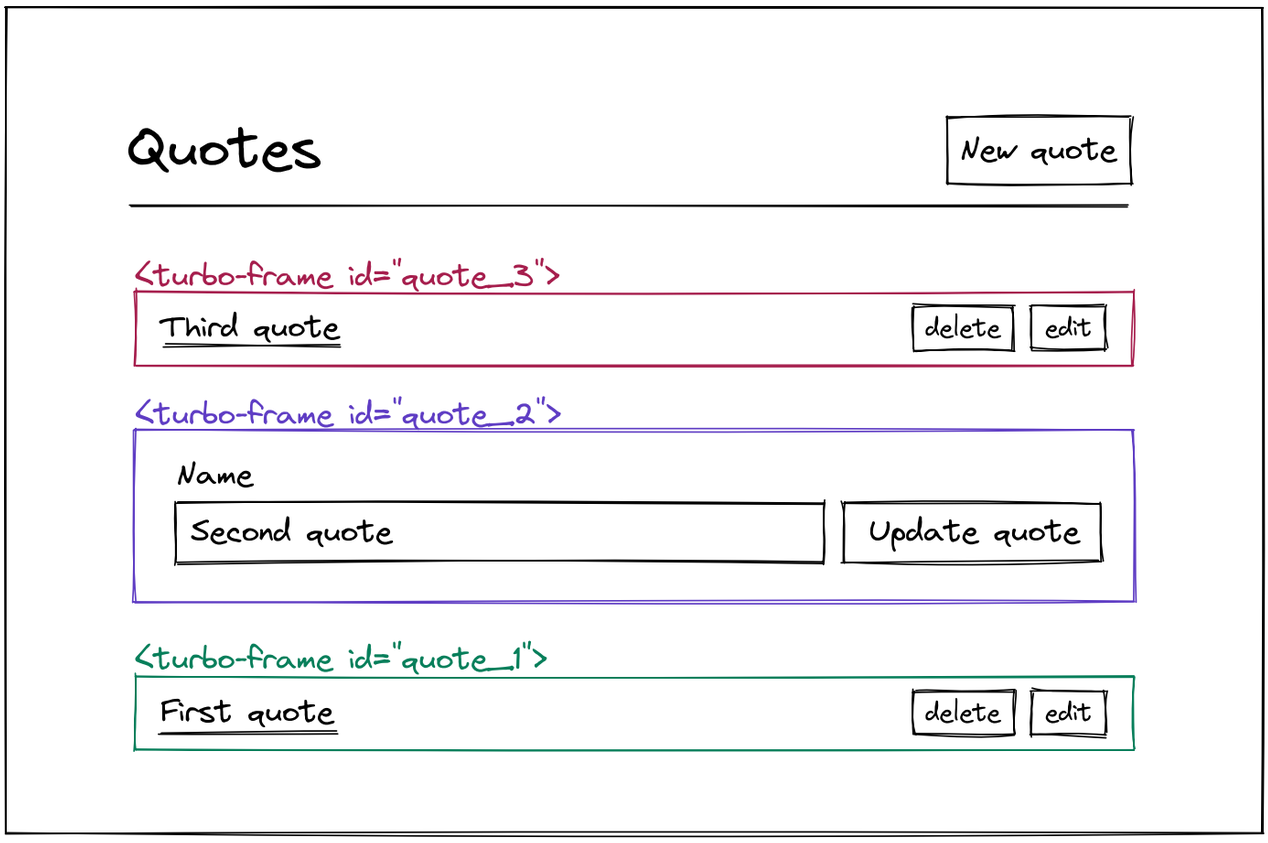
Sketch of the Quotes#index page with Turbo Frames around each quote

Now let's suppose we want to edit the **second quote**. When clicking on the "Edit" button of the **second quote**, we need a Turbo Frame with the same id on the `Quotes#edit` page as described on the following sketch:



Sketch of the Quotes#edit page with the Turbo Frame around the form

With our Turbo Frames appropriately named, when clicking on the "Edit" button of the second quote on the `Quotes#index` page, the content of the Turbo Frame containing the form should replace the content of the Turbo Frame containing the second quote card like described in the following sketch:



Sketch of the Quotes#index page with form to edit the second quote

With those sketches in mind and the rules of the previous section, let's implement this behavior. On the `Quotes#index` page, let's wrap each quote in a Turbo Frame with an id of `"quote_#{quote_id}"`. As each quote card on the

`Quotes#index` page is rendered from the `_quote.html.erb` partial, we simply need to wrap each quote within a Turbo Frame with this id:

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

<%= turbo_frame_tag "quote_#{quote.id}" do %>
  <div class="quote">
    <%= link_to quote.name, quote_path(quote) %>
    <div class="quote__actions">
      <%= button_to "Delete",
                    quote_path(quote),
                    method: :delete,
                    class: "btn btn--light" %>
      <%= link_to "Edit",
                  edit_quote_path(quote),
                  class: "btn btn--light" %>
    </div>
  </div>
<% end %>
```

We need a Turbo Frame of the same id around the form of the `Quotes#edit` page:

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

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>Edit quote</h1>
  </div>

  <%= turbo_frame_tag "quote_#{@quote.id}" do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

Now with only those four lines of code added, let's try our code in the browser. Let's click on the "Edit" button for a quote. The form successfully replaces the quote card.

Let's submit the form to see if it works as expected.

First, let's submit an **invalid blank form**:

1. When clicking on the "Update quote" button, Turbo intercepts the submit event as the form is nested within a Turbo Frame.
2. The form submission is invalid , so the controller renders the `app/quotes/edit.html.erb` view with the errors on the form.
3. Thanks to the 422 response status added by the `status: :unprocessable_entity` option, Turbo knows it has to replace the content of the Turbo Frame with the new one containing errors.
4. The errors are successfully displayed on the page.

Let's now submit a **valid form**:

1. When clicking on the "Update quote" button, Turbo intercepts the submit event as the form is nested within a Turbo Frame.
2. The form submission is valid on the controller side, so the controller redirects to the `Quotes#index` page.
3. The updated `Quotes#index` page contains a Turbo Frame of the same id that contains a card with the updated quote name.
4. Turbo replaces the frame's content containing the form with the frame's content containing the updated quote card.

Everything now works as expected for the `#edit` and `#update` actions.

However, you might notice some unexpected behavior. Clicking on the link to show a quote does not work anymore, and you might see an error in the console when deleting a quote. We will talk about this very soon (it has to do with rule number 2!), but before we move on, let's talk about the `dom_id` helper that will help us write cleaner Turbo Frame ids.

Turbo Frames and the dom_id helper

There is one more thing to know about the `turbo_frame_tag` helper. You can pass it a string or any object that can be converted to a `dom_id`. The `dom_id` helper helps us convert an object into a unique id like this:

```
# If the quote is persisted and its id is 1:
dom_id(@quote) # => "quote_1"
```

```
# If the quote is a new record:
dom_id(Quote.new) # => "new_quote"

# Note that the dom_id can also take an optional prefix argument
# We will use this later in the tutorial
dom_id(Quote.new, "prefix") # "prefix_new_quote"
```

The `turbo_frame_tag` helper automatically passes the given object to `dom_id`. Therefore, we can refactor our two `turbo_frame_tag` calls in our `Quotes#index` and `Quotes#edit` views by passing an object instead of a string. The following blocks of code are equivalent:

```
<%= turbo_frame_tag "quote_#{@quote.id}" do %>
  ...
<% end %>

<%= turbo_frame_tag dom_id(@quote) do %>
  ...
<% end %>

<%= turbo_frame_tag @quote %>
  ...
<% end %>
```

Let's refactor the code we just wrote to use this *syntactic sugar*:

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

<%= turbo_frame_tag quote do %>
  <div class="quote">
    <%= link_to quote.name, quote_path(quote) %>
    <div class="quote__actions">
      <%= button_to "Delete",
                    quote_path(quote),
                    method: :delete,
                    class: "btn btn--light" %>
      <%= link_to "Edit",
                  edit_quote_path(quote),
                  class: "btn btn--light" %>
    </div>
  </div>
<% end %>
```

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

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>Edit quote</h1>
  </div>

  <%= turbo_frame_tag @quote do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

Now that our Turbo Frames have great names, we can continue our work.

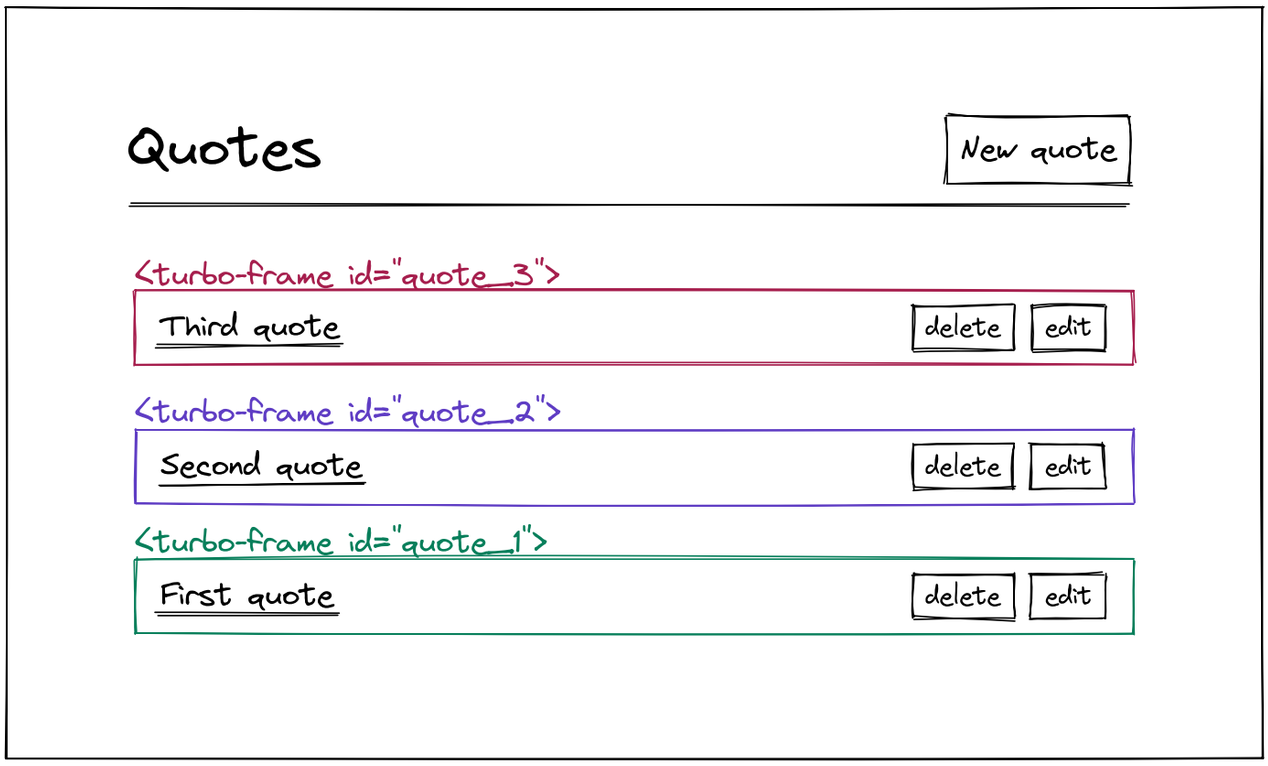
Showing and deleting quotes

Our implementation above works well for the `#edit` and `#update` actions, but we introduced two new issues:

1. The link to show a quote does not work as expected: the Turbo Frame containing the quote disappears, and an error is logged in the console.
2. The button to destroy a quote logs an error in the console.

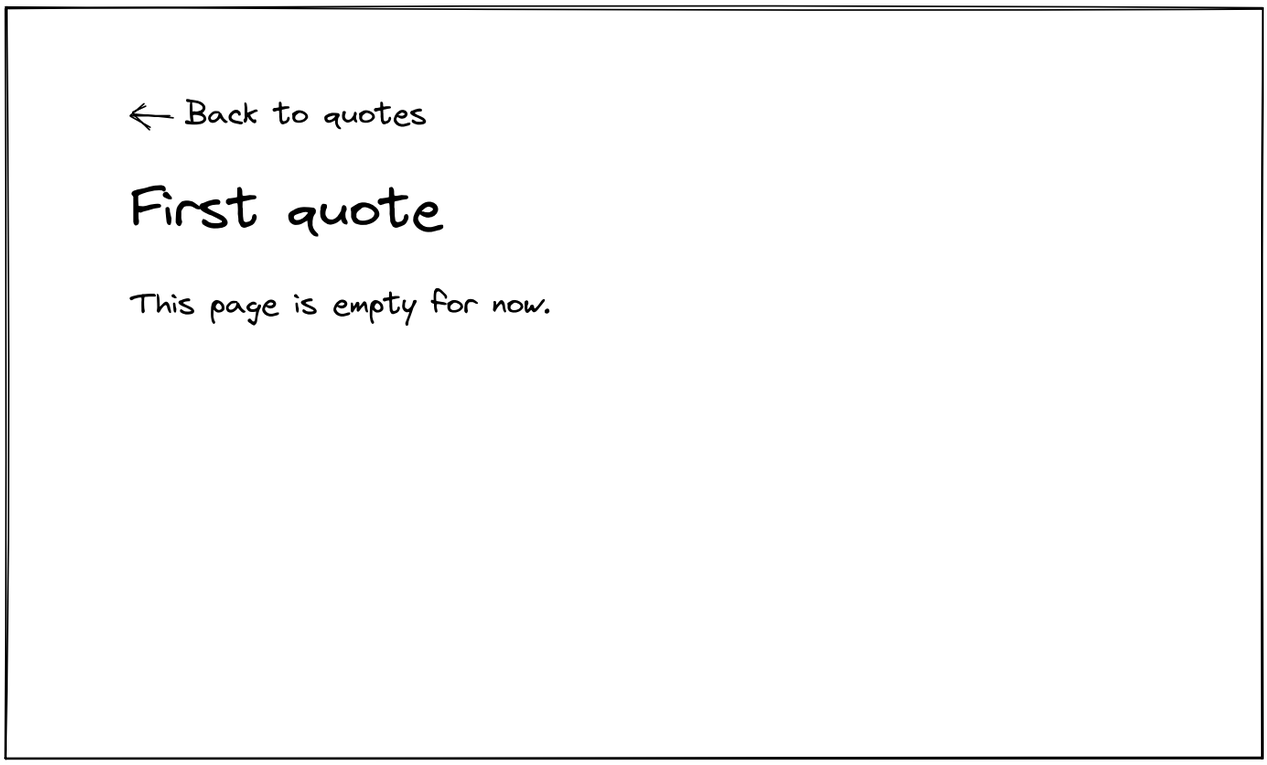
Both these issues have to do with rule number 2 of Turbo Frames we saw earlier. Let's solve them one by one.

Our `Quotes#index` page currently looks like this:



Sketch of the Quotes#index page with Turbo Frames around each quote

As we can see, the link to show a quote is nested within a Turbo Frame of id `"quote_#{quote.id}"`. However, on the `Quotes#show` page, there is no Turbo Frame with the same id:



Sketch of the Quotes#show page

Turbo expects a Turbo Frame of the same id on the `Quotes#show` page. To solve the problem, we will make the links to the `Quote#show` page target the `"_top"` frame to replace the whole page:

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

```
<%= turbo_frame_tag quote do %>
  <div class="quote">
    <%= link_to quote.name,
                quote_path(quote),
                data: { turbo_frame: "_top" } %>
    <div class="quote__actions">
      <%= button_to "Delete",
                    quote_path(quote),
                    method: :delete,
                    class: "btn btn--light" %>
      <%= link_to "Edit",
                  edit_quote_path(quote),
                  class: "btn btn--light" %>
    </div>
  </div>
<% end %>
```

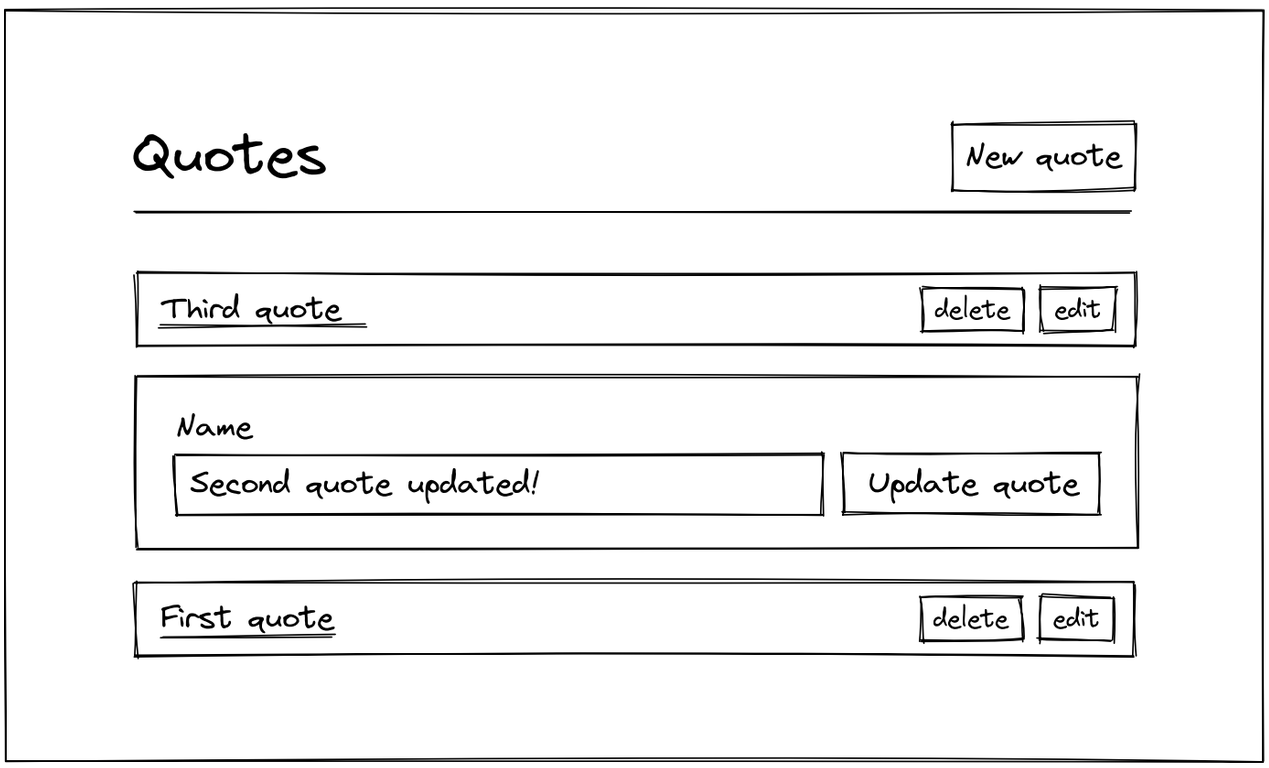
Let's test it in the browser. Our first problem is solved. Our links to the `Quotes#show` page now work as expected!

We *could* solve the second problem with the same method by making the form to delete the quote target the "_top" frame:

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

<%= turbo_frame_tag quote do %>
  <div class="quote">
    <%= link_to quote.name,
                quote_path(quote),
                data: { turbo_frame: "_top" } %>
    <div class="quote__actions">
      <%= button_to "Delete",
                    quote_path(quote),
                    method: :delete,
                    form: { data: { turbo_frame: "_top" } },
                    class: "btn btn--light" %>
      <%= link_to "Edit",
                  edit_quote_path(quote),
                  class: "btn btn--light" %>
    </div>
  </div>
<% end %>
```

If we test in the browser, it works as expected! There is no more error in the console!

While this is a perfectly valid solution, it has an unintended side effect we might want to address. Imagine if we open the form for the **second quote**, and click on the "Delete" button for the **third quote** like in this example:



Sketch of the Quotes#index page with a form to edit the second quote

Go ahead and test it in the browser. The **third quote** is removed as expected, but the response also closes the form for the **second quote**. This is because, as the form to delete the **third quote** targets the `"_top"` frame, the whole page is replaced!

It would be nice if we could only remove the Turbo Frame containing the deleted quote and leave the rest of the page unchanged to preserve the *state* of the page. Well, Turbo and Rails once again have our back! Let's remove what we just did for the "Delete" button:

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

<%= turbo_frame_tag quote do %>
  <div class="quote">
    <%= link_to quote.name,
                quote_path(quote),
                data: { turbo_frame: "_top" } %>
    <div class="quote__actions">
      <%= button_to "Delete",
```

```
                    quote_path(quote),
                    method: :delete,
                    class: "btn btn--light" %>
      <%= link_to "Edit",
                  edit_quote_path(quote),
                  class: "btn btn--light" %>
    </div>
  </div>
<% end %>
```

It's time for an introduction to the `TURBO_STREAM` format.

The Turbo Stream format

Forms in Rails 7 are now submitted with the `TURBO_STREAM` format. Let's destroy a quote and inspect what happens in the log of our Rails server:

```
Started DELETE "/quotes/908005781" for 127.0.0.1 at 2022-01-27 15:30:13
Processing by QuotesController#destroy as TURBO_STREAM
```

As we can see, the `QuotesController` will process the `#destroy` action with the `TURBO_STREAM` format. Let's explore what we can do with this format by making our destroy action only remove the Turbo Frame containing the deleted quote while leaving the rest of the page untouched.

In the controller, let's support both the `HTML` and the `TURBO_STREAM` formats thanks to the `respond_to` method:

```
# app/controllers/quotes_controller.rb

def destroy
  @quote.destroy

  respond_to do |format|
    format.html { redirect_to quotes_path, notice: "Quote was successful
    format.turbo_stream
  end
end
```

As with any other format, let's create the corresponding view:

```
<%# app/views/quotes/destroy.turbo_stream.erb %>

<%= turbo_stream.remove "quote_#{@quote.id}" %>
```

Let's delete a quote and inspect the response body in the "Network" tab in the browser. The HTML received by the browser should look like this, except you probably have a different id for the quote:

```
<turbo-stream action="remove" target="quote_908005780">
</turbo-stream>
```

Where does this HTML come from? In the `TURBO_STREAM` view we just created, the `turbo_stream` helper received the `remove` method with the `"quote_#{@quote.id}"` as an argument. As we can see, this helper converts this into a `<turbo-stream>` custom element with the action "remove" and the target "quote_908005780".

When the browser receives this HTML, Turbo will know how to interpret it. It will perform the desired **action** on the Turbo Frame with the id specified by the **target** attribute. In our case, Turbo *removes* the Turbo Frame corresponding to the deleted quote leaving the rest of the page untouched. That's exactly what we wanted!

Note: As of writing this chapter, the `turbo_stream` helper responds to the following methods, so that it can perform the following actions:

```
# Remove a Turbo Frame
turbo_stream.remove

# Insert a Turbo Frame at the beginning/end of a list
turbo_stream.append
turbo_stream.prepend

# Insert a Turbo Frame before/after another Turbo Frame
turbo_stream.before
turbo_stream.after

# Replace or update the content of a Turbo Frame
turbo_stream.update
turbo_stream.replace
```

Of course, except for the `remove` method, the `turbo_stream` helper expects a *partial* and *locals* as arguments to know which HTML it needs to *append*, *prepend*, *replace* from the DOM. In the next section, we will learn how to pass *partials* and *locals* to the `turbo_stream` helper.

With the combination of Turbo Frames and the new `TURBO_STREAM` format, we will be able to perform precise operations on pieces of our web pages without having to write a single line of JavaScript, therefore preserving the *state* of our web pages.

One last thing before we move on to the next section, the `turbo_stream` helper can also be used with `dom_id`. We can refactor our view like this:

```
<%# app/views/quotes/destroy.turbo_stream.erb %>

<%= turbo_stream.remove @quote %>
```

It's time to tackle the last feature: our quote creation.

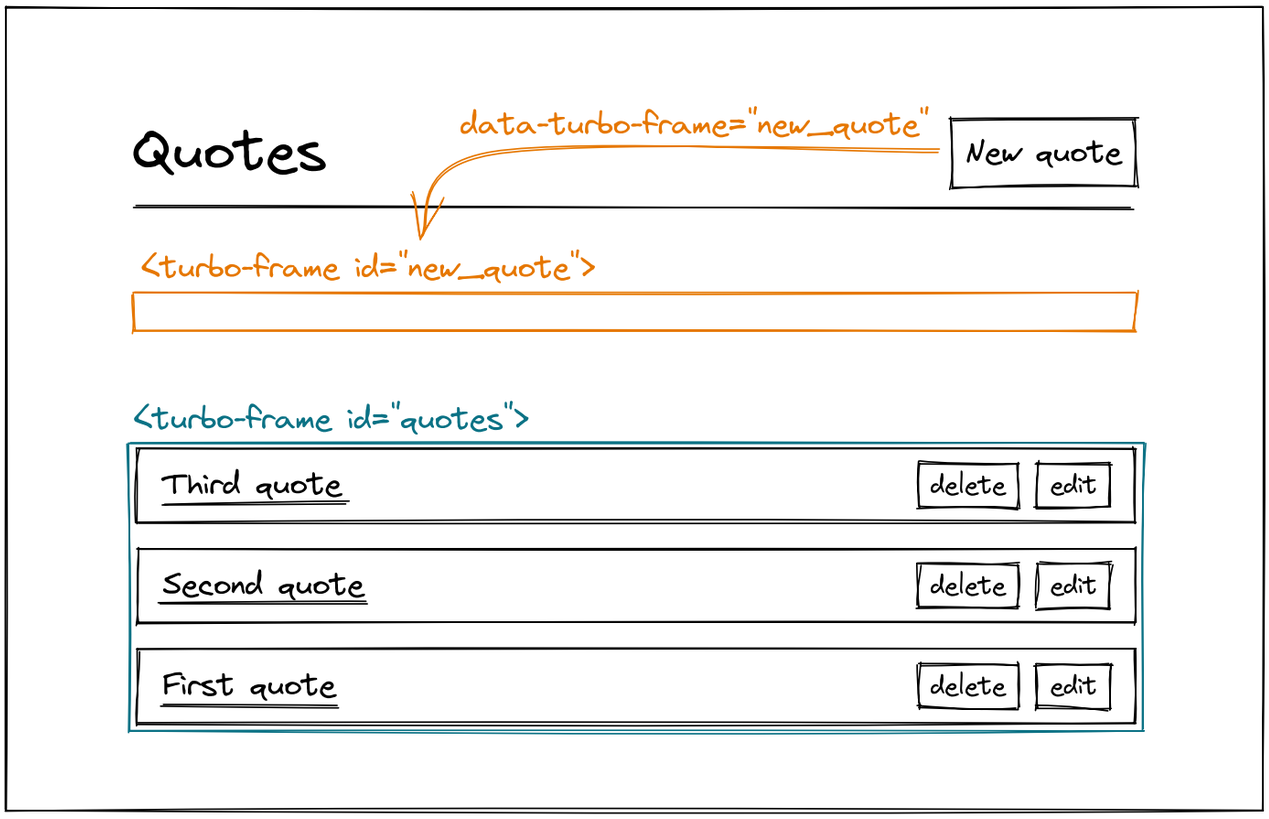
Creating a new quote with Turbo Frames

The last feature we need is the quote creation. Before diving into the implementation, let's draw a few sketches of what we will build.

Clicking on the "New quote" button won't bring us to the `Quotes#new` page anymore. Instead, the new quote form will appear right below the header on the `Quotes#index` page. Then, clicking on the "Create quote" button will *prepend* the newly created quote to the list and remove the new quote form from the page. To do this, we will need two more Turbo Frames:

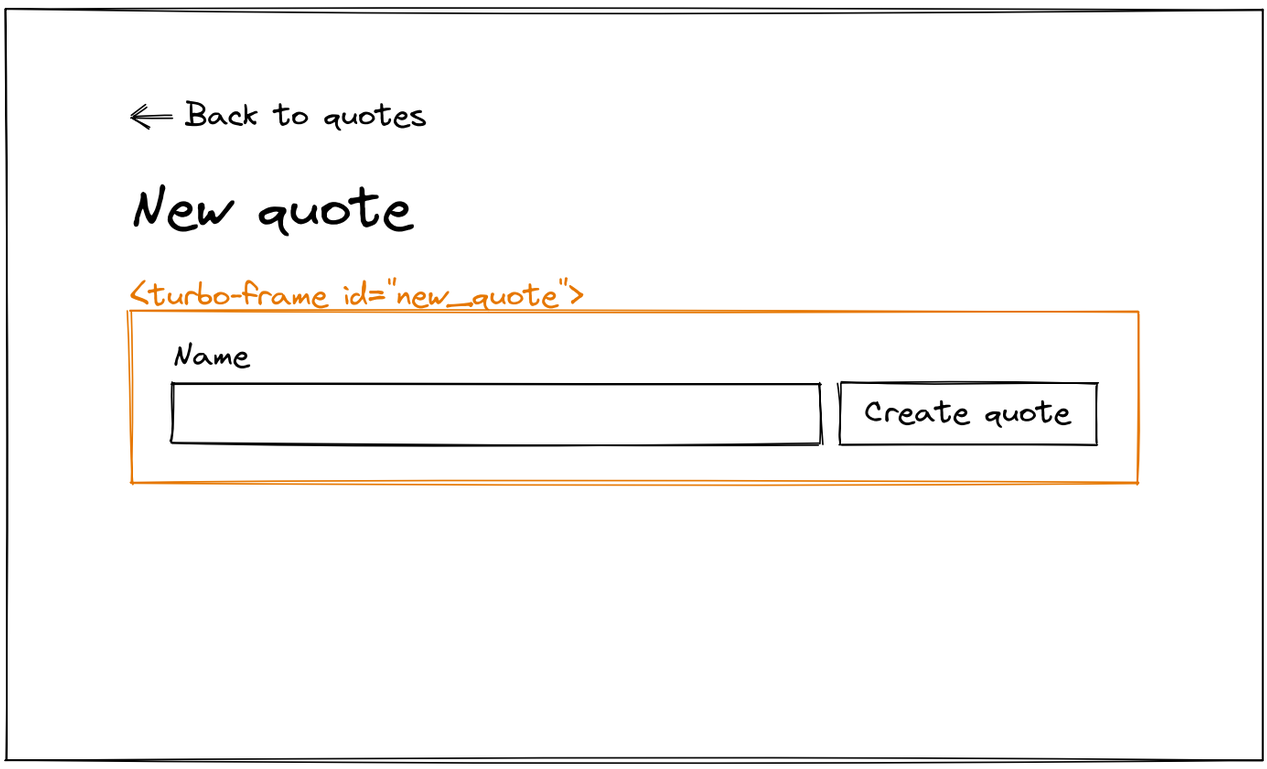
- One empty Turbo Frame that will receive the new quote form.
- One Turbo Frame that wraps the quotes list for us to *prepend* the newly created quote at the correct position.

This is described in the sketch below:



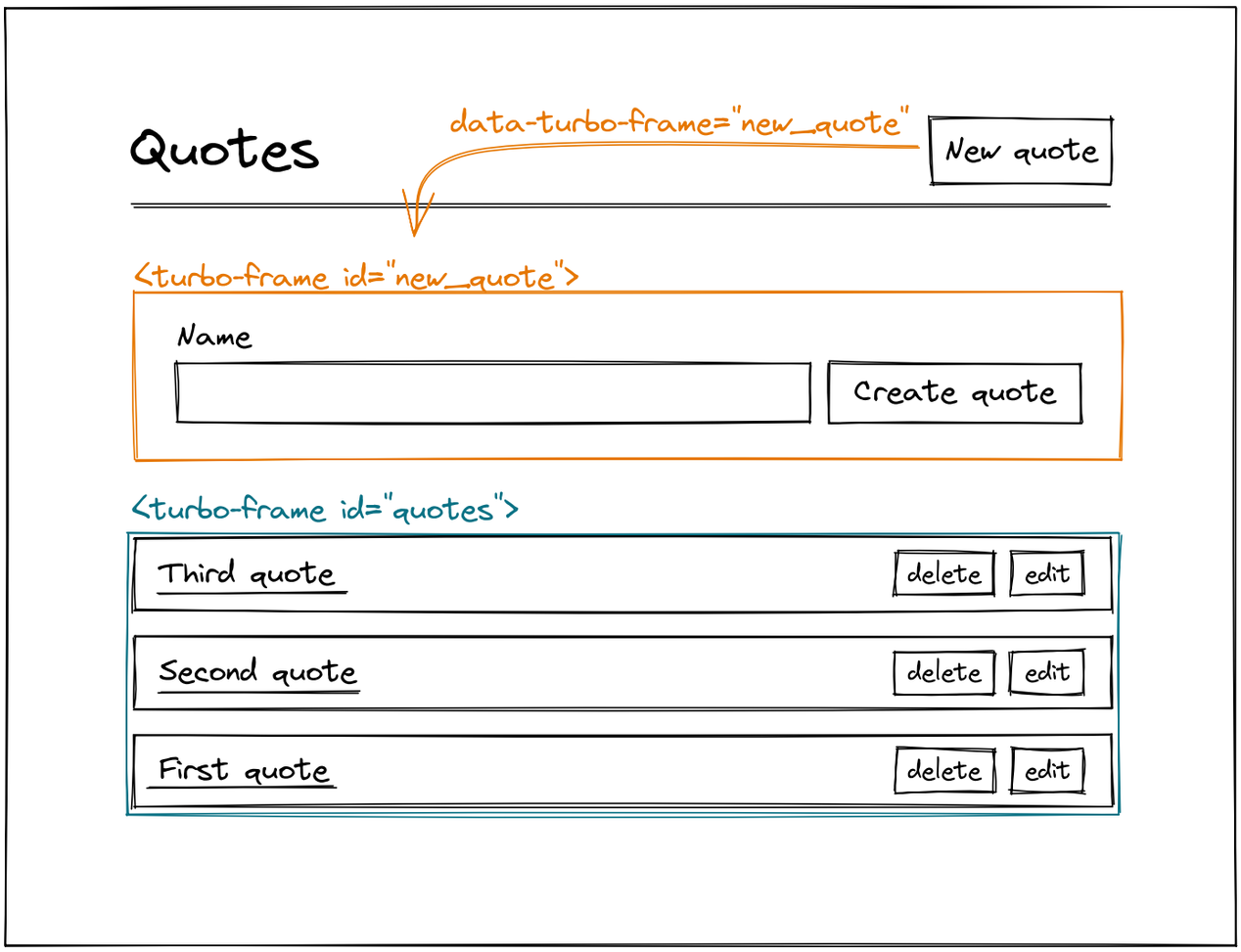
Sketch of the Quotes#index page

On the `Quotes#new` page, we will wrap the form in a Turbo Frame with the same id as the empty one on the `Quotes#index` page:



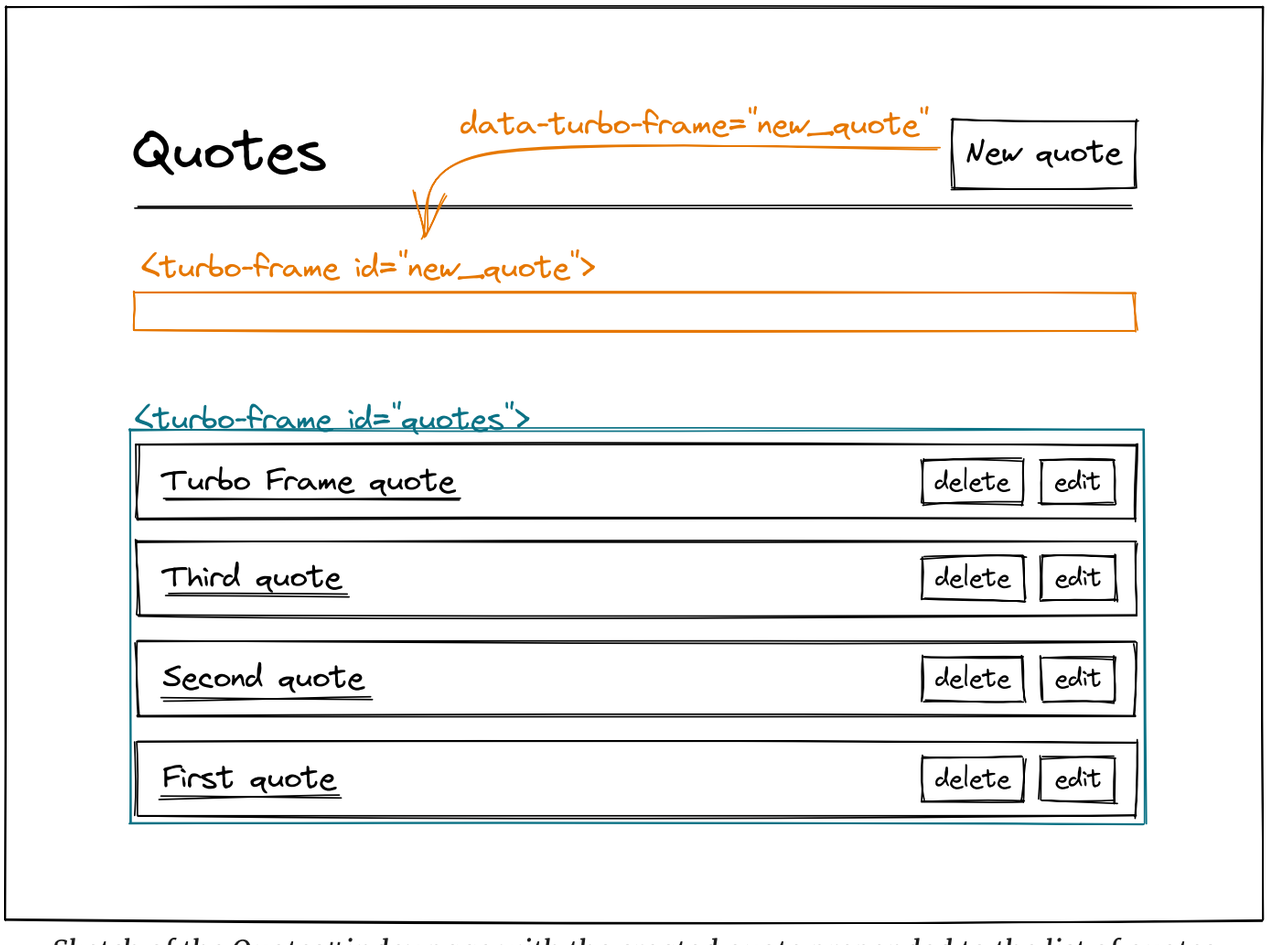
Sketch of the Quotes#new page with the form wrapped into a Turbo Frame

With our Turbo Frames appropriately named, clicking on the "New quote" button will replace the empty content of the `"new_quote"` frame with the form to create a new quote:



Sketch of the Quotes#index page with the new quote form below the header

When submitting the form, we will leverage the power of Turbo Streams to *prepend* the new quote to the list of quotes and *update* the `"new_quote"` frame to be empty again:



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

Let's implement our solution.

As mentioned earlier, we need two frames of the same id on the `Quotes#index` view and the `Quotes#new` page. Those frames will have the id of `dom_id(Quote.new)`, which is equivalent to the string `"new_quote"`.

On the `Quotes#new` page, let's wrap the new quote form in the Turbo Frame:

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

<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>

  <div class="header">
    <h1>New quote</h1>
  </div>

  <%= turbo_frame_tag @quote do %>
    <%= render "form", quote: @quote %>
  <% end %>
</main>
```

Here `@quote` is a new record, so the three following expressions are equivalent:

```
turbo_frame_tag "new_quote"
turbo_frame_tag Quote.new
turbo_frame_tag @quote
```

Now let's add an empty Turbo Frame of the same id to the `Quotes#index` page that will receive this new quote form:

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

<main class="container">
  <div class="header">
    <h1>Quotes</h1>
    <%= link_to "New quote",
                new_quote_path,
                class: "btn btn--primary",
                data: { turbo_frame: dom_id(Quote.new) } %>
  </div>
```

```
  <%= turbo_frame_tag Quote.new %>
  <%= render @quotes %>
</main>
```

As you can see, the frame is empty on the `Quotes#index` page. The new quote form should only appear when clicking on the "New quote" button. To link our "New quote" button with the correct Turbo Frame we just created, we have to use the `data-turbo-frame` data attribute:

As we can see, the `data-turbo-frame` attribute on the link matches the id of the empty Turbo Frame, thus connecting the two. Let's break down what happens here:

1. When clicking on the "New quote" link, the click will be intercepted by Turbo.
2. Turbo knows it has to interact with the frame of id `new_quote` thanks to the attribute `data-turbo-frame` on the "New quote" link.
3. The request is sent in AJAX, and our server will render the `Quotes#new` page with a frame with id `new_quote`.
4. When the browser receives the HTML, Turbo will extract the frame with the id of `new_quote` from the `Quotes#new` page and replace the empty frame with the same id on the `Quotes#index` page!

Let's test it in our browser. It works! Our form appears on the page as expected!

Let's now try to submit the form **with a blank name** by clicking on the "Create Quote" button. The quote will be invalid, and the errors should appear on the page. Let's explain what happens here:

1. When clicking on the "Create Quote" button, the form submission is intercepted by Turbo.
2. The form is wrapped in a frame with id `new_quote`, so Turbo knows it only needs to replace this frame.
3. The server receives the invalid params in the `QuotesController#create` action and renders the `Quotes#new` view with the form containing errors.
4. When the browser receives the response with the `status: :unprocessable_entity`, it replaces the frame with id `new_quote` with the new one that contains errors.

We are almost there. To complete the feature as designed in our sketch, we need to prepend the newly created quote to the list of quotes when a valid name is given to the quote.

If we test it now in the browser, we will realize that the quote gets created in the database, but the created quote does not get prepended to the list of quotes.

Why is that?

When submitting the form with valid attributes, the `QuotesController#create` action will render the `Quotes#index` page that contains an empty frame of id `new_quote` that will replace our form. However, Turbo does not know what to do with the newly created quote. Where should it be inserted on the page? Should it be appended to a list of quotes? Or maybe prepended? To do this, we will use a Turbo Stream view!

Let's tell the `QuotesController` that it needs to support both the `HTML` and the `TURBO_STREAM` formats:

```
# app/controllers/quotes_controller.rb

def create
  @quote = Quote.new(quote_params)

  if @quote.save
    respond_to do |format|
      format.html { redirect_to quotes_path, notice: "Quote was successf
      format.turbo_stream
    end
  else
    render :new, status: :unprocessable_entity
  end
end
```

Let's create the corresponding view:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", partial: "quotes/quote", locals: { qu
<%= turbo_stream.update Quote.new, "" %>
```

In this view, we instruct Turbo to do two things:

1. The first line tells Turbo to *prepend* to the Turbo Frame with id `quotes` the `app/views/quotes/_quote.html.erb` partial. As we can see, it's straightforward to pass a *partial* and *locals* to the `turbo_stream` helper.
2. The second line tells Turbo to update the Turbo Frame with id `new_quote` with empty content.

The last thing we need to do to make it work is adding a Turbo Frame with id "quotes" to wrap the list of quotes in the `Quotes#index` page.

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

<div class="container">
  <div class="header">
    <h1>Quotes</h1>
    <%= link_to "New quote",
                new_quote_path,
                class: "btn btn--primary",
                data: { turbo_frame: dom_id(Quote.new) } %>
  </div>

  <%= turbo_frame_tag Quote.new %>

  <%= turbo_frame_tag "quotes" do %>
    <%= render @quotes %>
  <% end %>
</div>
```

Let's now test it out in the browser. It works! If we inspect the "Network" tab in the dev tools when submitting a valid new quote form, the response body should look like this:

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

<turbo-stream action="update" target="new_quote">
```

```
  <template>
    <!-- An empty template! -->
  </template>
</turbo-stream>
```

As you can see, it matches our two lines in the `create.turbo_stream.erb` view we just created translated in a language Turbo can understand! When receiving the response, Turbo executes the `action` (append, prepend, replace, remove) on the `target` Turbo Frame.

Note: There are different ways to write the same things in Turbo Stream views. Let's look at our `create.turbo_stream.erb` view we just created.

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", partial: "quotes/quote", locals: { qu
<%= turbo_stream.update Quote.new, "" %>
```

While this is a perfectly valid way of writing our view, there is another syntax with a block I sometimes use when lines are too long:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes" do %>
  <%= render partial: "quotes/quote", locals: { quote: @quote } %>
<% end %>

<%= turbo_stream.update Quote.new, "" %>
```

In Ruby on Rails, the following two expressions are equivalent:

```
render partial: "quotes/quote", locals: { quote: @quote }
render @quote
```

With this in mind, we can again shorten the way we write our view:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes" do %>
  <%= render @quote %>
```

```
<% end %>

<%= turbo_stream.update Quote.new, "" %>
```

We don't need the block syntax here as the lines are short, so this will be the final way we write the view:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", @quote %>
<%= turbo_stream.update Quote.new, "" %>
```

Elegant right? This note was just a way to learn the different ways to write the same Turbo Stream views!

Ordering our quotes

There is one last detail we should take care of. We decided to prepend the created quote to the list of quotes, but when we refresh the page, the order of the quotes in the list changes. To always keep the quotes ordered the newest first, let's add a scope to our `Quote` model:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  validates :name, presence: true

  scope :ordered, -> { order(id: :desc) }
end
```

Let's then use this scope in our controller in the `#index` action:

```
# app/controllers/quotes_controller.rb

def index
  @quotes = Quote.ordered
end
```

Now the order of quotes is consistent even when we refresh the page. That's a small detail, but it might be important for our users to understand what happens.

This breaks our system test as the ordering isn't the same anymore so let's update our system test to make them pass again:

```
# test/system/quotes_test.rb

setup do
  # We need to order quote as well in the system tests
  @quote = Quote.ordered.first
end
```

Let's run our tests; they all should be green! Our quote editor now looks exactly as described in the first sketches. We had to learn some new skills, but the implementation was only a few lines of code. Turbo is an incredible piece of software!

Adding a cancel button

Now that everything works as expected, let's add the last improvement to our page. **We want our users to be able to close new/edit quote forms without submitting them**. To do this, we will add a "Cancel" link that links to the `Quotes#index` page on the `quotes/_form.html.erb` partial:

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

<%= simple_form_for quote, html: { class: "quote form" } do |f| %>
  <% if quote.errors.any? %>
    <div class="error-message">
      <%= quote.errors.full_messages.to_sentence.capitalize %>
    </div>
  <% end %>

  <%= f.input :name, input_html: { autofocus: true } %>
  <%= link_to "Cancel", quotes_path, class: "btn btn--light" %>
  <%= f.submit class: "btn btn--secondary" %>
<% end %>
```

Let's test it in the browser. Thanks to the power of Turbo Frames, it already works! Let's explain what happens.

When our user clicks on the "Cancel" link for the **new quote form**:

1. The link is within a Turbo Frame of id `new_quote`, so Turbo will only replace the content of this frame
2. The link navigates to the `Quotes#index` page that contains an empty Turbo Frame with id `new_quote`
3. Turbo replaces the content of the `new_quote` frame with the empty content, so the form disappears

When our user clicks on the "Cancel" link for the **edit quote form**:

1. The link is within a Turbo Frame of id `dom_id(quote)`, so Turbo will only replace the content of this frame
2. The link navigates to the `Quotes#index` page that contains a Turbo Frame with id `dom_id(quote)` that contains the HTML for this quote
3. Turbo replaces the content of the `dom_id(quote)` Frame containing the form with the HTML for this quote

If we try to create a quote and edit multiple quotes simultaneously, we will notice that the *state* of the page is preserved. For example, when creating a quote, all the edition forms that are opened will remain open. Turbo Frames are independent pieces of the web page that we can manipulate without writing any custom JavaScript!

Wrap up

In this chapter, we replaced our classic CRUD controller on the quotes resource with a modern *reactive* application with almost no code and no JavaScript!

Let's take some time to play with what we have built. Let's open a form to edit a quote, destroy another quote, click on the "New quote" button. Compared to using a frontend library like React, there is no state to manage, no complex actions to dispatch, no reducers... Turbo is a joy to work with!

This was a dense chapter so let's take a break and make sure everything is clear in our heads before we start the next one!

In the next chapter, we will talk about making real-time updates in our application by broadcasting Turbo Streams with Action Cable. See you there!

← previous next →