← Volver a la lista de capítulos

Un controlador CRUD simple con Rails

Publicado el 5 de enero de 2022

En este primer capítulo, comenzaremos nuestra aplicación creando nuestro modelo de cotización y su controlador asociado siguiendo las convenciones de Ruby on Rails.

El CRUD con Ruby on Rails

Comenzaremos nuestro viaje creando un controlador CRUD simple para nuestro `Quote` modelo.

Lo bueno de Turbo es que podemos empezar a crear nuestra aplicación exactamente como lo haríamos *sin* él. **Si seguimos las convenciones de Ruby on Rails, podremos hacer que nuestra aplicación sea reactiva agregando solo unas pocas líneas de código y sin escribir ningún código JavaScript** .

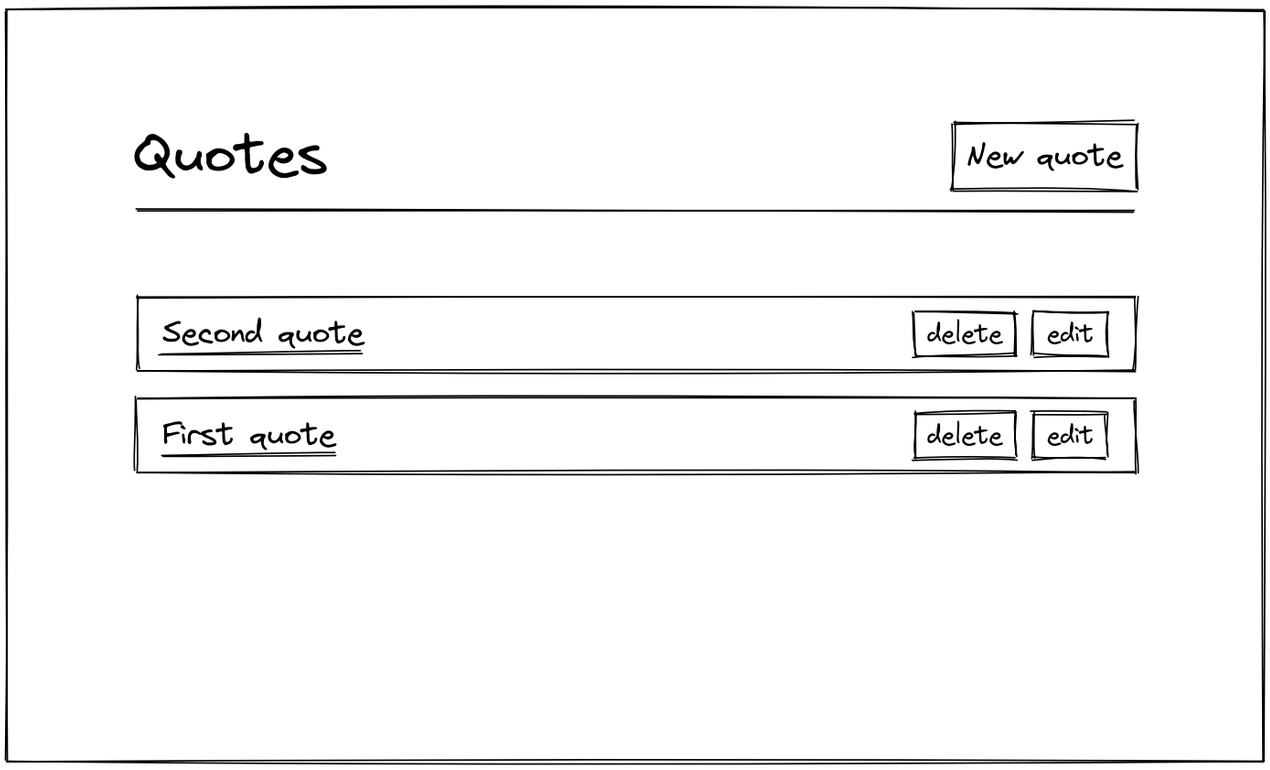
Para asegurarnos de que estamos en la misma página, primero creemos algunos bocetos de lo que construiremos en este capítulo con la fantástica herramienta de diseño gratuita Excalidraw .

En la `Quotes#index` página, mostraremos una lista de todas las cotizaciones. Para cada cotización, tendremos tres componentes:

- Un enlace para mostrar la cita.
- Un botón para editar la cotización.

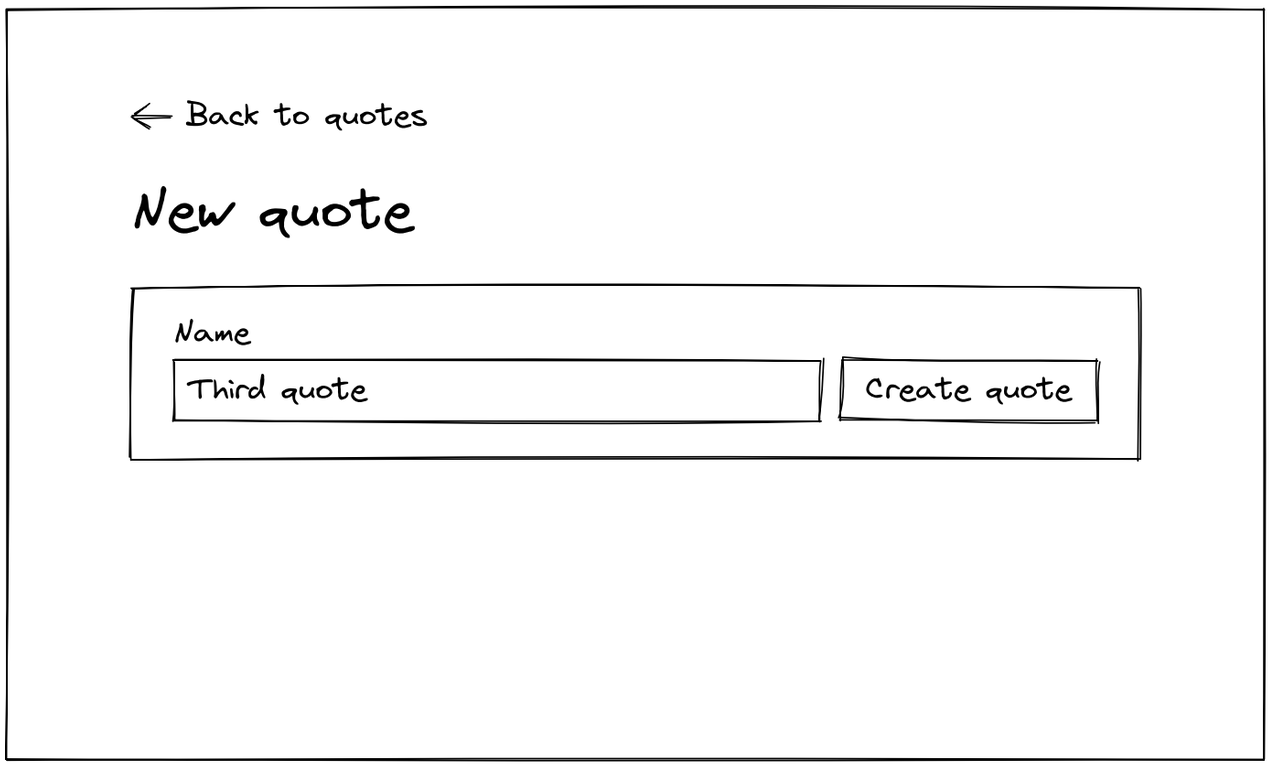
- Un botón para destruir la cita.

La `Quotes#index` página también tendrá un botón "Nueva cita" para agregar nuevas citas a la lista de citas como se describe en el siguiente esquema:



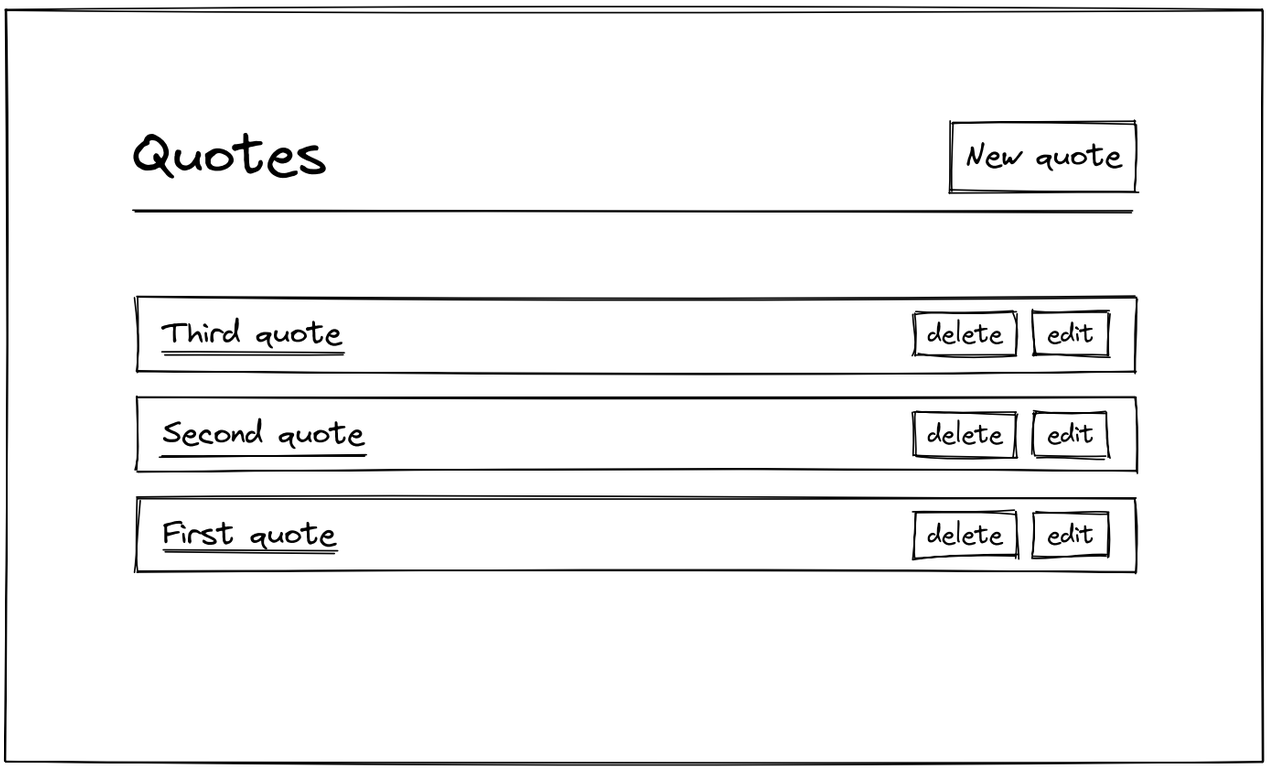
Boceto de la página de citas#índice

Al hacer clic en el botón “Nueva cotización” accederemos a la `Quotes#new` página donde podremos rellenar un formulario para crear una nueva cotización.



Boceto de las citas#nueva pagina

Al hacer clic en el botón “Crear cotización” nos redireccionará a la `Quotes#index` página, donde veremos la nueva cotización agregada a la lista de cotizaciones. Tenga en cuenta que las cotizaciones están ordenadas de la más reciente en la parte superior de la lista.

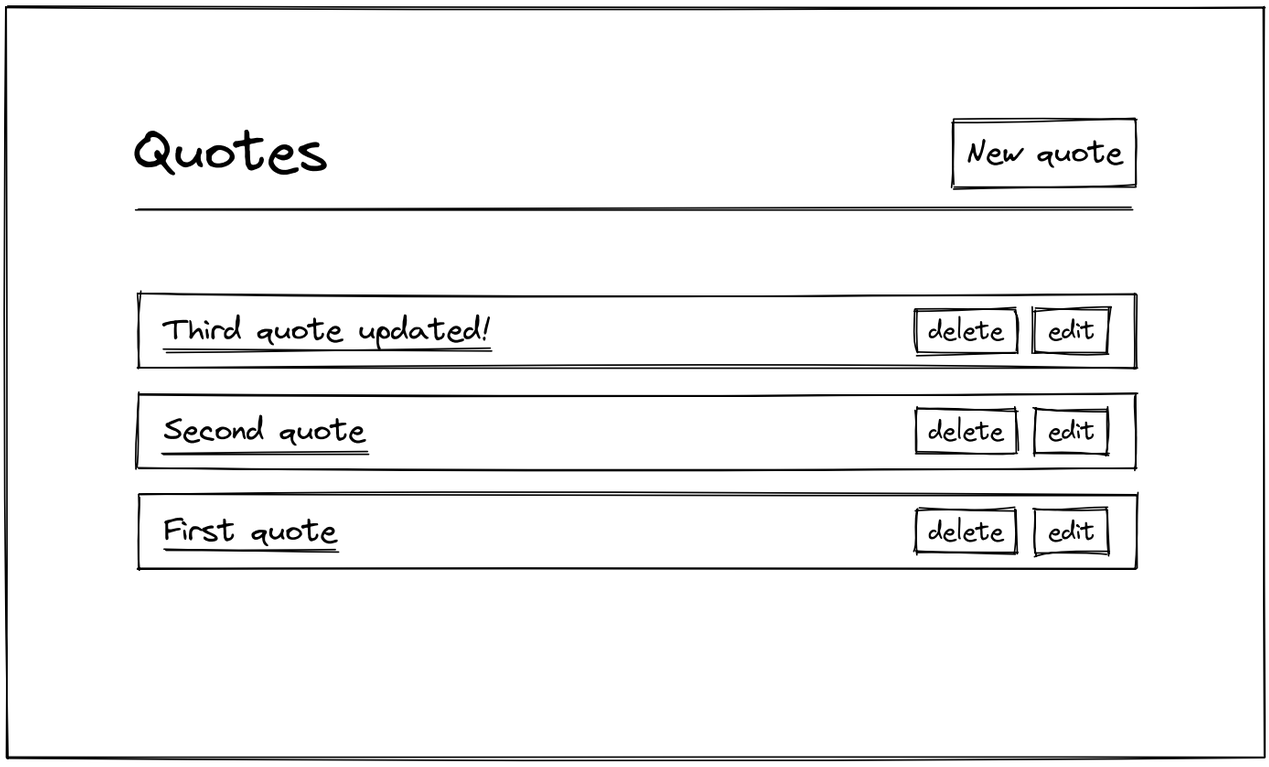


Boceto de la página de índice de citas con la cita creada

Al hacer clic en el botón "Editar" en una cotización específica nos llevará a la `Quotes#edit` página para actualizar una cotización existente.

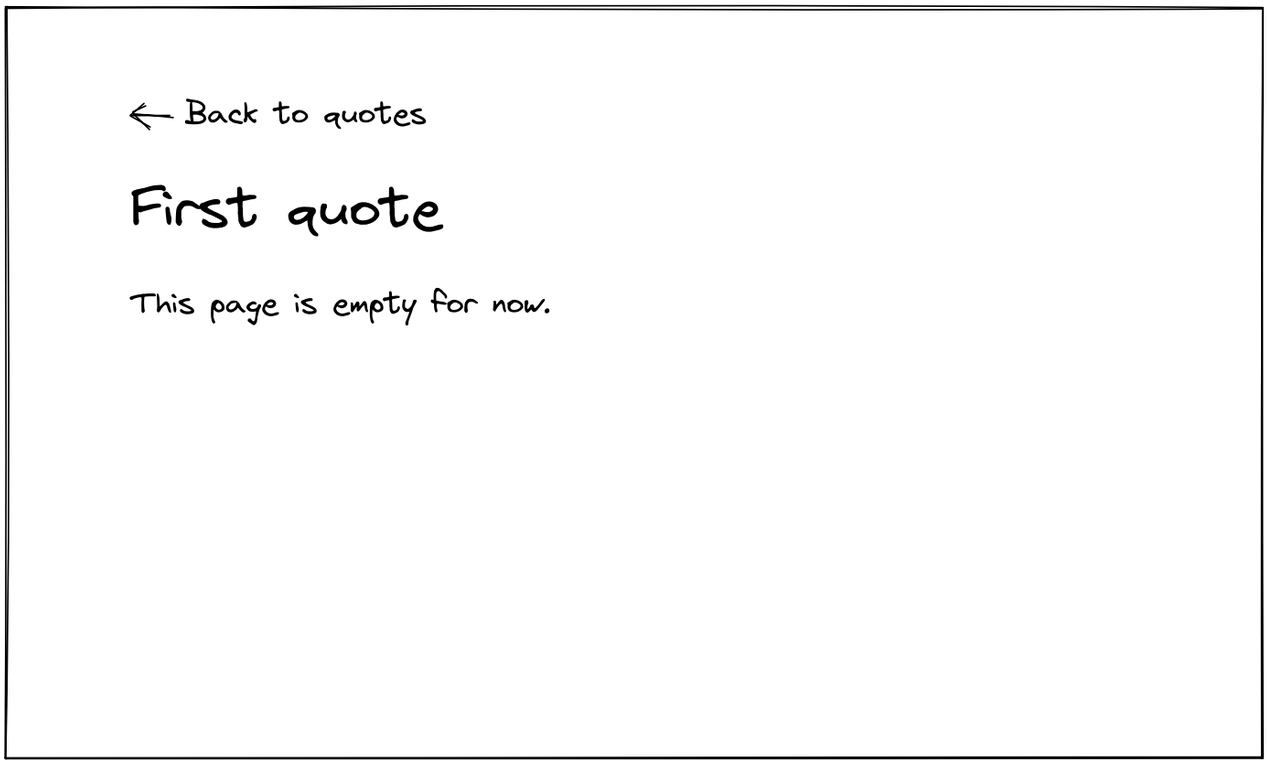
Boceto de la página de citas#edit

Al hacer clic en el botón “Actualizar cotización” nos redireccionará a la `Quotes#index` página donde podremos ver nuestra cotización actualizada.



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

Por último, pero no por ello menos importante, al hacer clic en el nombre de una cita, accederemos a la `Quotes#show` página. Por ahora, solo contendrá el título de la cita, pero más adelante agregaremos muchas más funciones a esta página.



Boceto de la página vacía de Quotes#show

¿Listo para comenzar? Primero, agreguemos algunas pruebas para asegurarnos de que lo que estamos creando funcionará como se espera.

Probando nuestra aplicación Rails

Las pruebas son una parte fundamental del desarrollo de software . Sin un conjunto de pruebas sólido, introduciremos errores en nuestra aplicación y romperemos funciones que solían funcionar sin siquiera saberlo. Haremos que nuestro trabajo sea más tedioso y queremos evitarlo.

En esta aplicación, escribiremos pruebas del sistema Rails para asegurarnos de que nuestra aplicación siempre se comporte como queremos. Afortunadamente, esas pruebas son fáciles de escribir.

Primero ejecutemos el generador para crear el archivo de prueba para nosotros:

```
bin/rails g system_test quotes
```

Con la ayuda de los bocetos anteriores, describamos lo que sucede en un lenguaje sencillo y escribamos algunas pruebas al mismo tiempo:

```
# test/system/quotes_test.rb

require "application_system_test_case"

class QuotesTest < ApplicationSystemTestCase
  test "Creating a new quote" do
    # When we visit the Quotes#index page
    # we expect to see a title with the text "Quotes"
    visit quotes_path
    assert_selector "h1", text: "Quotes"

    # When we click on the link with the text "New quote"
    # we expect to land on a page with the title "New quote"
    click_on "New quote"
    assert_selector "h1", text: "New quote"

    # When we fill in the name input with "Capybara quote"
    # and we click on "Create Quote"
    fill_in "Name", with: "Capybara quote"
    click_on "Create quote"

    # We expect to be back on the page with the title "Quotes"
    # and to see our "Capybara quote" added to the list
    assert_selector "h1", text: "Quotes"
    assert_text "Capybara quote"
```

```
  end
end
```

¿Ves cómo esa prueba se lee como si estuviera en lenguaje sencillo? ¡Tenemos mucha suerte de trabajar con Ruby on Rails! Necesitamos tres pruebas más para mostrar, actualizar y destruir una cita, pero necesitamos algunos datos de prueba para escribirlas.

En este tutorial se utilizarán los accesorios de Rails para crear datos falsos para nuestras pruebas. Los accesorios vienen con Rails de forma predeterminada y son muy fáciles de usar. Consisten en un archivo YAML para cada modelo. Cada entrada en los archivos de accesorios YAML se cargará en la base de datos antes de que se ejecute el conjunto de pruebas.

Primero, crearemos el archivo de accesorios para nuestras cotizaciones:

```
touch test/fixtures/quotes.yml
```

Vamos a crear algunas citas en este archivo:

```
# test/fixtures/quotes.yml

first:
  name: First quote

second:
  name: Second quote

third:
  name: Third quote
```

Ahora estamos listos para agregar dos pruebas más a nuestro conjunto de pruebas:

```
# test/system/quotes_test.rb

require "application_system_test_case"

class QuotesTest < ApplicationSystemTestCase
  setup do
    @quote = quotes(:first) # Reference to the first fixture quote
```

```
  end

  # ...
  # The test we just wrote
  # ...

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

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

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

    click_on "Edit", match: :first
    assert_selector "h1", text: "Edit quote"

    fill_in "Name", with: "Updated quote"
    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
```

Ahora que nuestras pruebas están listas, podemos ejecutarlas con `bin/rails test:system`. Como podemos observar, todas fallan porque nos faltan rutas, un `Quote` modelo y un `QuotesController`. Ahora que nuestros requisitos son precisos, es hora de comenzar a trabajar en la esencia de nuestra aplicación.

Creando nuestro modelo de cotización

Primero, creemos el `Quote` modelo con un `name` atributo y su archivo de migración asociado ejecutando el siguiente comando en la consola. **Como ya generamos el archivo de accesorios, escriba "n" para "no" cuando se le solicite que lo anule** :

```
rails generate model Quote name:string
```

Todas nuestras cotizaciones deben tener un nombre para ser válidas, por lo que agregaremos esto como validación en el modelo:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  validates :name, presence: true
end
```

En la `CreateQuotes` migración, agreguemos `null: false` una restricción a nuestro `name` atributo para aplicar la validación y garantizar que nunca almacenaremos comillas con un nombre vacío en la base de datos, incluso si cometemos un error en la consola.

```
# db/migrate/XXXXXXXXXXXXXX_create_quotes.rb

class CreateQuotes < ActiveRecord::Migration[7.0]
  def change
    create_table :quotes do |t|
      t.string :name, null: false

      t.timestamps
    end
  end
end
```

Ahora estamos listos para ejecutar la migración:

```
bin/rails db:migrate
```

Añadiendo nuestras rutas y acciones del controlador

Ahora que nuestro modelo está listo, es momento de desarrollar nuestro controlador CRUD. Vamos a crear el archivo con la ayuda del generador de Rails:

```
bin/rails generate controller Quotes
```

Agreguemos las siete rutas del CRUD para nuestro `Quote` recurso:

```
# config/routes.rb

Rails.application.routes.draw do
  resources :quotes
end
```

Ahora que todas las rutas están configuradas, podemos escribir las acciones del controlador correspondientes:

```
# app/controllers/quotes_controller.rb

class QuotesController < ApplicationController
  before_action :set_quote, only: [:show, :edit, :update, :destroy]

  def index
    @quotes = Quote.all
  end

  def show
  end

  def new
    @quote = Quote.new
  end

  def create
    @quote = Quote.new(quote_params)

    if @quote.save
      redirect_to quotes_path, notice: "Quote was successfully created."
    else
      render :new
    end
  end
```

```
  def edit
  end

  def update
    if @quote.update(quote_params)
      redirect_to quotes_path, notice: "Quote was successfully updated."
    else
      render :edit
    end
  end

  def destroy
    @quote.destroy
    redirect_to quotes_path, notice: "Quote was successfully destroyed."
  end

  private

  def set_quote
    @quote = Quote.find(params[:id])
  end

  def quote_params
    params.require(:quote).permit(:name)
  end
end
```

¡Genial! Lo último que tenemos que hacer para que nuestras pruebas pasen es crear las vistas.

Nota : vamos muy rápido. Si tienes problemas con este controlador, debes empezar leyendo la guía de inicio de la documentación de Ruby on Rails y volver al tutorial cuando termines de leerlo.

Añadiendo nuestras vistas de cotizaciones

Nota : Las vistas que agregaremos ya tienen algunos nombres de clase CSS. Construiremos nuestro sistema de diseño y crearemos los archivos CSS correspondientes en el próximo capítulo, por lo que, por ahora, puede simplemente copiar el marcado. Esto es para mayor comodidad, de modo que no tengamos que volver y editar esas vistas en el próximo capítulo.

El marcado para la página Quotes#index

El primer archivo que tenemos que crear es la `Quotes#index` página `app/views/quotes/index.html.erb`.

```
<%# 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>
```

Seguimos las convenciones de Ruby on Rails para la `Quotes#index` página al representar cada cita en la `@quotes` colección desde el archivo parcial `app/views/quotes/_quote.html.erb`. Por eso necesitamos este segundo archivo:

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

<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>
```

El marcado para las páginas Quotes#new y Quotes#edit

Los primeros archivos que tenemos que crear son `app/views/quotes/new.html.erb` y `app/views/quotes/edit.html.erb`.

Tenga en cuenta que la única diferencia entre las dos páginas es el contenido de la `<h1>` etiqueta.

```
<%# 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>
```

```
<%# 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>

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

Una vez más, seguiremos las convenciones de Ruby on Rails al representar el formulario desde el `app/views/quotes/_form.html.erb` archivo. De esa manera, podemos usar el mismo parcial tanto para las páginas `Quotes#new` como para las `Quotes#edit` páginas.

```
<%# 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 } %>
```

```
  <%= f.submit class: "btn btn--secondary" %>
<% end %>
```

La opción de enfoque automático está aquí para enfocar el campo de entrada correspondiente cuando el formulario aparece en la pantalla, de modo que no tengamos que usar el mouse y podamos escribir directamente en él. ¿Observa cómo el marcado del formulario es simple? Esto se debe a que vamos a usar la `simple_form` gema. Para instalarla, agreguemos la gema a nuestro archivo `Gemfile`.

```
# Gemfile

gem "simple_form", "~> 5.1.0"
```

Con nuestra gema agregada, es hora de instalarla:

```
bundle install
bin/rails generate simple_form:install
```

La función de la `simple_form` gema es facilitar el trabajo con los formularios. También ayuda a mantener la coherencia de los diseños de los formularios en toda la aplicación, ya que garantiza que siempre usemos las mismas clases CSS. Reemplacemos el contenido del archivo de configuración y desglosémoslo:

```
# config/initializers/simple_form.rb

SimpleForm.setup do |config|
  # Wrappers configration
  config.wrappers :default, class: "form__group" do |b|
    b.use :html5
    b.use :placeholder
    b.use :label, class: "visually-hidden"
    b.use :input, class: "form__input", error_class: "form__input--inval
  end

  # Default configuration
  config.generate_additional_classes_for = []
  config.default_wrapper                 = :default
  config.button_class                    = "btn"
  config.label_text                      = lambda { |label, _, _| label
  config.error_notification_tag          = :div
  config.error_notification_class        = "error_notification"
```

```
  config.browser_validations             = false
  config.boolean_style                   = :nested
  config.boolean_label_class             = "form__checkbox-label"
end
```

La *configuración de los wrappers* enumera los wrappers de entrada que podemos utilizar en nuestra aplicación. Por ejemplo, si llamamos `f.input :name` al `Quote` modelo con el `:default` wrapper, se generará el siguiente código HTML:

```
<div class="form__group">
  <label class="visually-hidden" for="quote_name">
    Name
  </label>
  <input class="form__input" type="text" name="quote[name]" id="quote_na
</div>
```

¿Puedes ver cómo las clases CSS del HTML generado coinciden con la definición de nuestro contenedor? ¡Eso es lo que hace Simple Form! Nos ayuda a definir contenedores para formularios que podemos reutilizar en nuestra aplicación muy fácilmente, lo que hace que sea más fácil trabajar con nuestros formularios.

La parte *de configuración predeterminada* contiene opciones para configurar las clases predeterminadas de los botones de envío, las etiquetas, la forma en que se representan las casillas de verificación y los botones de opción... La más importante aquí es la `config.default_wrapper = :default`. Significa que cuando escribimos `f.input :name` sin especificar un contenedor, `:default` se utilizará el contenedor que se ve arriba para generar el HTML.

Resulta que solo necesitaremos el `:default` contenedor en toda nuestra aplicación, por lo que hemos terminado con la `simple_form` configuración de la gema.

El formulario simple también nos ayuda a definir texto para etiquetas y marcadores de posición en otro archivo de configuración:

```
# config/locales/simple_form.en.yml

en:
```

```
  simple_form:
    placeholders:
      quote:
        name: Name of your quote
    labels:
      quote:
        name: Name

  helpers:
    submit:
      quote:
        create: Create quote
        update: Update quote
```

La configuración anterior indica de forma simple que el marcador de posición para las entradas de nombres en los formularios de cotización debe ser "Nombre de su cotización" y que la etiqueta para las entradas de nombres de cotización debe ser "Nombre".

Las últimas líneas debajo de la `:helpers` clave se pueden usar sin la gema de formulario simple, pero las agregaremos de todos modos porque están relacionadas con los formularios. Definen el texto de los `submit` botones en los formularios de citas:

- Cuando la cotización sea un registro nuevo, el botón de enviar tendrá el texto "Crear cotización"
- Cuando la cotización ya está almacenada en la base de datos, el botón de envío tendrá el texto "Actualizar cotización".

El marcado para la página Quotes#show

La última vista que necesitamos es la `Quotes#show` página. Por ahora estará casi vacía, conteniendo únicamente un título con el nombre de la cita y un enlace para volver a la `Quotes#index` página.

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

<main class="container">
  <%= link_to sanitize("&larr; Back to quotes"), quotes_path %>
  <div class="header">
    <h1>
      <%= @quote.name %>
```

```
    </h1>
  </div>
</main>
```

Parece que acabamos de cumplir nuestra misión. Asegurémonos de que nuestra prueba sea exitosa ejecutando `bin/rails test:system`. ¡Pasaron!

Nota : Al iniciar las pruebas del sistema, veremos que se abre el navegador Google Chrome y se realizan las tareas que creamos para nuestra prueba del sistema de cotizaciones. Podemos usar el `headless_chrome` controlador en su lugar para evitar que se abra la ventana de Google Chrome:

```
# test/application_system_test_case.rb

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  # Change :chrome with :headless_chrome
  driven_by :selenium, using: :headless_chrome, screen_size: [1400, 1400
end
```

¡Con este simple cambio en la configuración, las pruebas ahora pasan sin abrir la ventana de Google Chrome cada vez!

También podemos iniciar nuestro servidor web con el `bin/dev` comando en la terminal y asegurarnos de que todo funciona correctamente. No olvidemos reiniciar nuestro servidor, ya que acabamos de instalar una nueva gema y modificamos un archivo de configuración.

Turbo Drive: Las respuestas del formulario deben redirigir a otra ubicación

La aplicación funciona como se espera **a menos que enviemos un formulario vacío** . Incluso si tenemos una validación de presencia en el nombre de la cita, el mensaje de error no se muestra como podríamos esperar cuando falla el envío del formulario. Si abrimos la consola en nuestras herramientas de desarrollo, veremos el críptico mensaje de error "Las respuestas del formulario deben redirigir a otra ubicación".

Este es un "cambio radical" desde Rails 7 debido a *Turbo Drive* . Analizaremos este tema en profundidad en el Capítulo 3 dedicado a *Turbo Drive* . Si alguna vez te encuentras con este problema, la forma de solucionarlo es agregar `status: :unprocessable_entity` a las acciones `QuotesController#create` y `QuotesController#update` cuando el formulario se envía con errores:

```
# app/controllers/quotes_controller.rb

class QuotesController < ApplicationController
  # ...

  def create
    @quote = Quote.new(quote_params)

    if @quote.save
      redirect_to quotes_path, notice: "Quote was successfully created."
    else
      # Add `status: :unprocessable_entity` here
      render :new, status: :unprocessable_entity
    end
  end

  # ...

  def update
    if @quote.update(quote_params)
      redirect_to quotes_path, notice: "Quote was successfully updated."
    else
      # Add `status: :unprocessable_entity` here
      render :edit, status: :unprocessable_entity
    end
  end

  # ...
end
```

Con el código de estado de respuesta agregado, nuestro CRUD ahora funciona perfectamente, pero la interfaz de usuario es realmente fea. En el próximo capítulo, solucionaremos este problema, donde crearemos un pequeño sistema de diseño para nuestro editor de cotizaciones (¡el que uso en este sitio web!).

Sembrando nuestra aplicación con datos de desarrollo

Cuando lanzamos nuestro servidor por primera vez, no teníamos ninguna cita en nuestra página. Sin datos de desarrollo, cada vez que queremos editar o eliminar una cita, debemos crearla primero.

Si bien esto está bien para una aplicación pequeña como esta, resulta molesto para las aplicaciones del mundo real. Imaginemos si tuviéramos que crear todos los datos manualmente cada vez que queremos agregar una nueva característica. Es por eso que la mayoría de las aplicaciones Rails tienen un script `db/seeds.rb` para completar la base de datos de desarrollo con datos falsos para ayudar a configurar datos realistas para el desarrollo.

Sin embargo, en nuestro tutorial ya tenemos accesorios para datos de prueba:

```
# test/fixtures/quotes.yml

first:
  name: First quote

second:
  name: Second quote

third:
  name: Third quote
```

Reutilizaremos los datos de los partidos para crear nuestros datos de desarrollo. Esto tendrá dos ventajas:

💎 Trenes calientes

`db/seeds.rb` archivos y en los archivos de partidos.
- Mantendremos sincronizados los datos de prueba y los datos de desarrollo.

Si te gusta usar accesorios para tus pruebas, es posible que sepas que, en lugar de ejecutar, `bin/rails db:seed` puedes ejecutar `bin/rails db:fixtures:load` para crear datos de desarrollo a partir de tus archivos de accesorios. Indiquémosle a Rails que los dos comandos son equivalentes en el `db/seeds.rb` archivo:

```
# db/seeds.rb

puts "\n== Seeding the database with fixtures =="
system("bin/rails db:fixtures:load")
```

Ahora, ejecutar el `bin/rails db:seed` comando equivale a eliminar todas las comillas y cargar los datos de desarrollo como datos de desarrollo. Cada vez que necesitemos restablecer datos de desarrollo limpios, podemos ejecutar el `bin/rails db:seed` comando:

```
bin/rails db:seed
```

Ahora que hemos configurado todo, podemos notar que el diseño está... ¡nada diseñado! En el próximo capítulo, escribiremos algunos CSS para crear los componentes que necesitamos para nuestro editor de cotizaciones. ¡Nos vemos allí!

← anterior Siguiente →