Hotrails

← Back to the list of chapters

A simple CRUD controller with Rails

Published on January 05, 2022

In this first chapter, we will start our application by creating our quote model and its associated controller following the Ruby on Rails conventions.

The CRUD with Ruby on Rails

We will begin our journey by creating a simple CRUD controller for our `Quote` model.

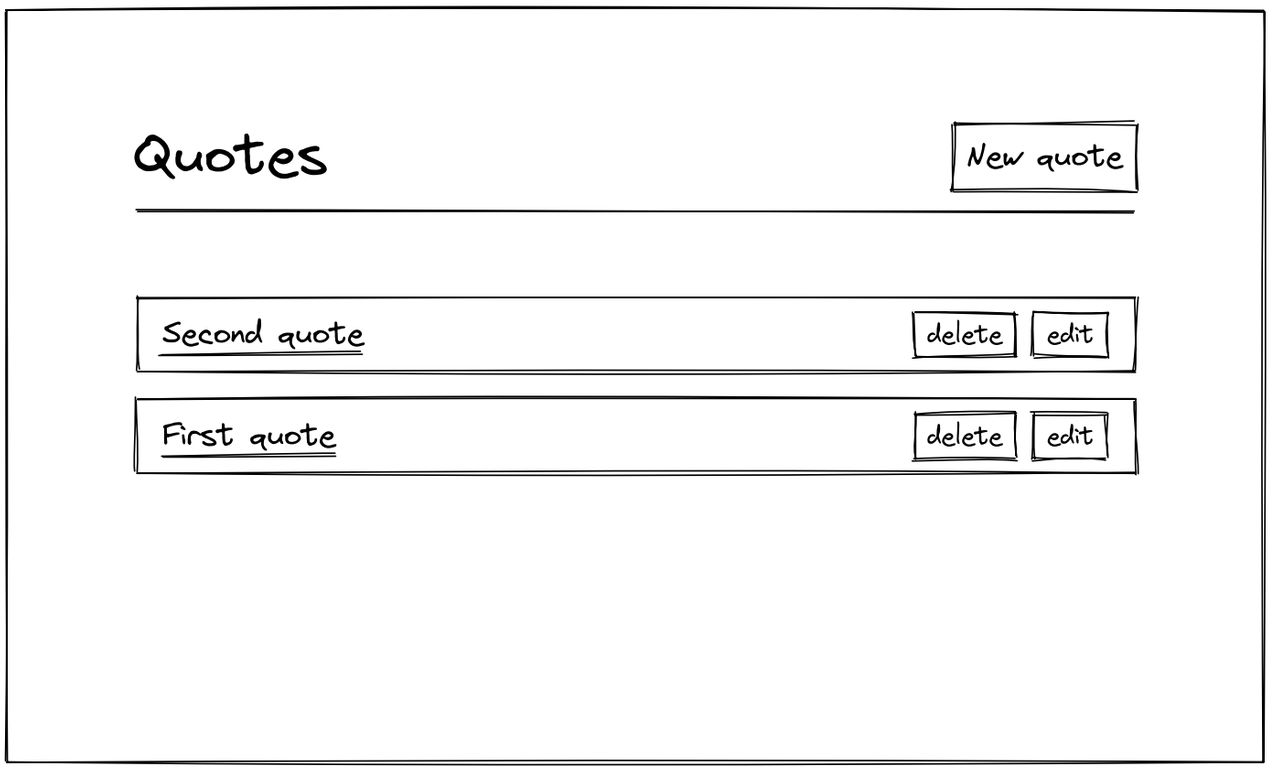
The great thing with Turbo is that we can start building our application exactly like we would *without* it. **If we follow the Ruby on Rails conventions, we will be able to make your application reactive by adding only a few lines of code and without writing any JavaScript**.

To ensure we are on the same page, let's first create a few sketches of what we will build in this chapter with the fantastic free design tool Excalidraw.

On the `Quotes#index` page, we will display a list of all the quotes. For each quote, we will have three components:

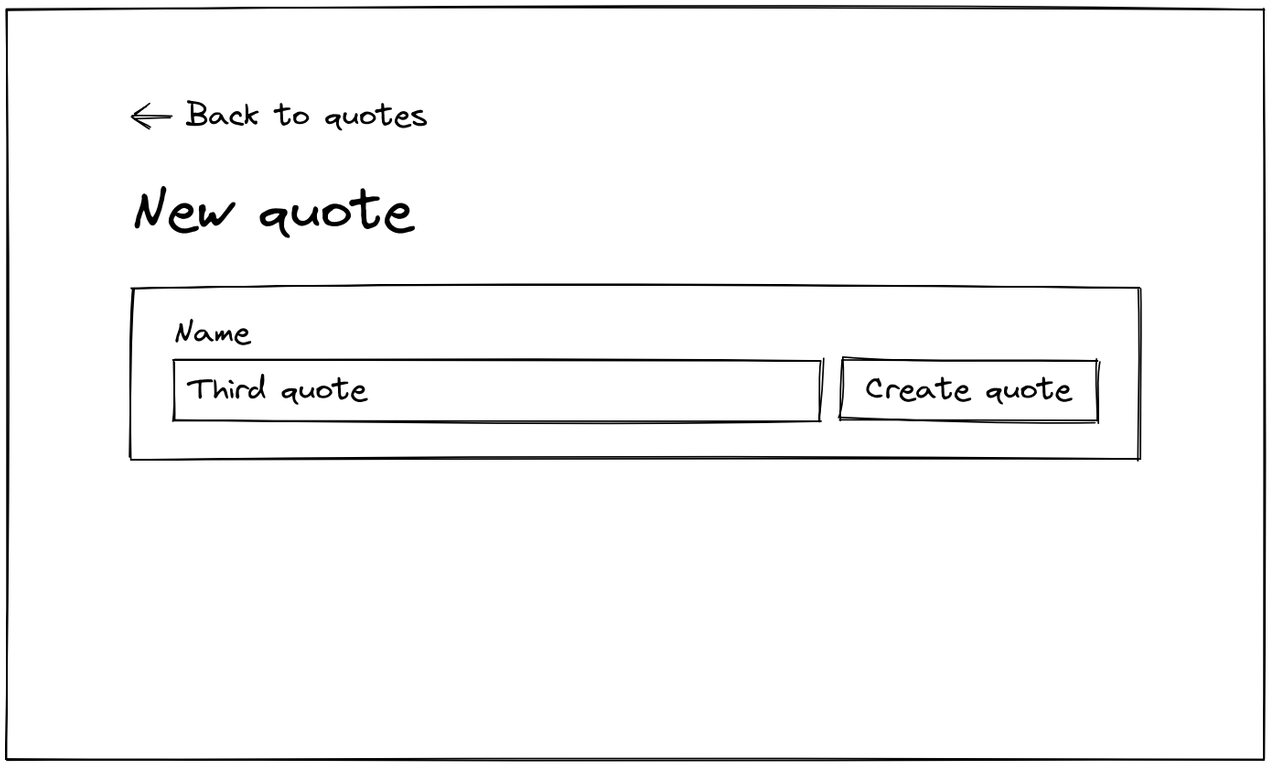
- A link to show the quote
- A button to edit the quote
- A button to destroy the quote

The `Quotes#index` page will also have a "New quote" button to add new quotes to the list of quotes as described in the following sketch:



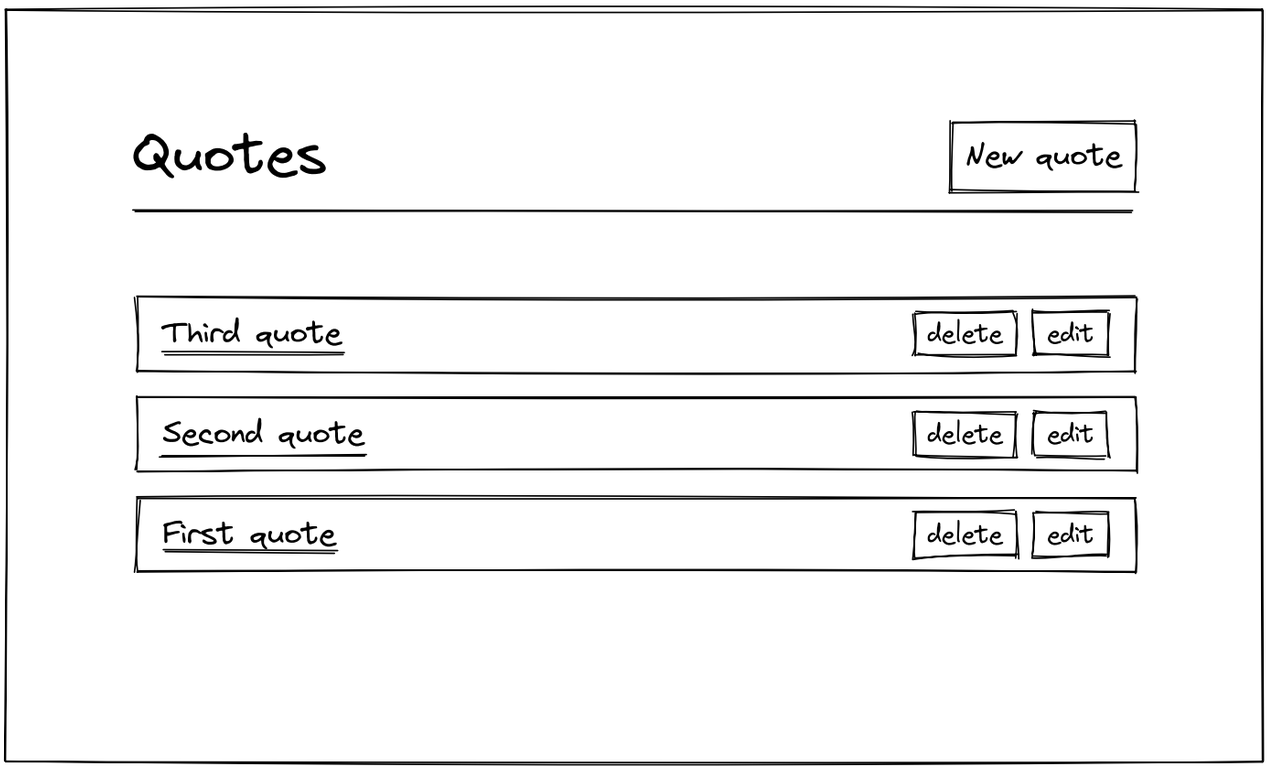
Sketch of the Quotes#index page

Clicking on the "New quote" button will take us to the `Quotes#new` page, where we can fill out a form to create a new quote.



Sketch of the Quotes#new page

Clicking on the "Create quote" button will redirect us to the `Quotes#index` page, where we will see the new quote added to the list of quotes. Note that the quotes are ordered the most recent at the top of the list.

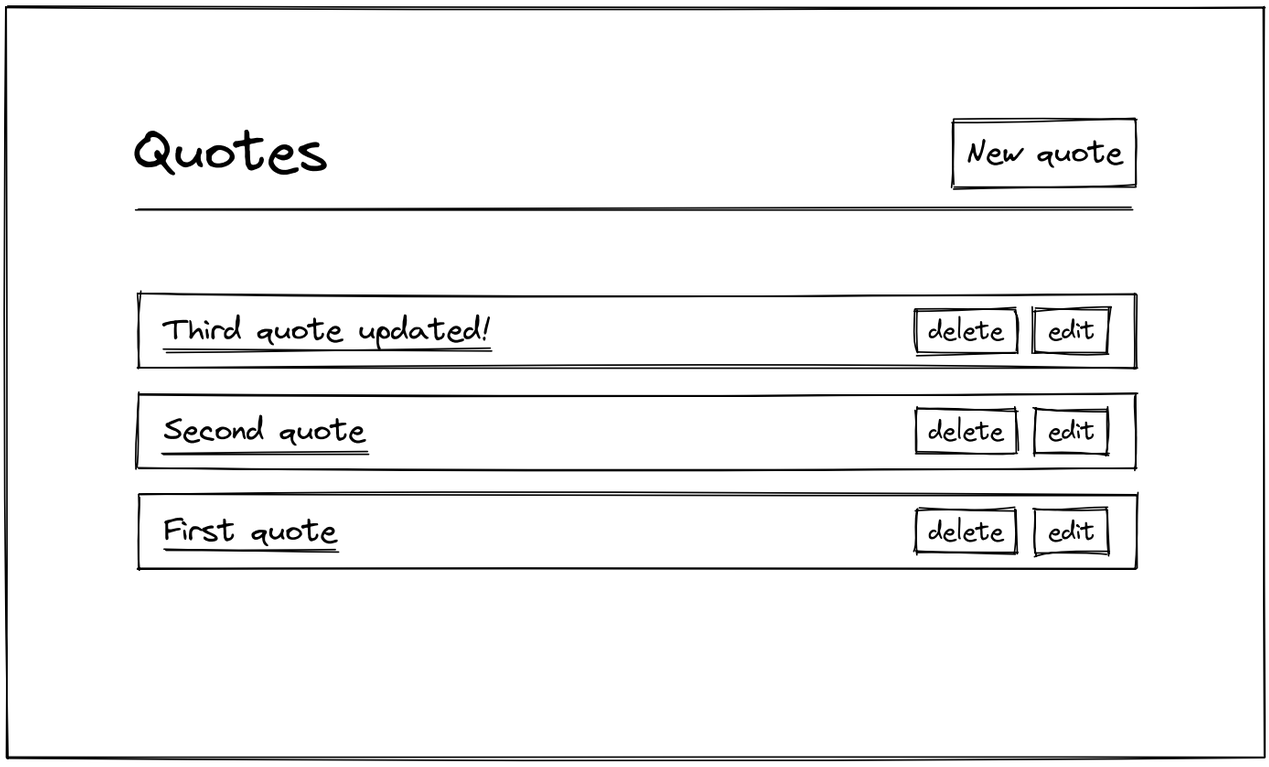


Sketch of the Quotes#index page with the created quote

Clicking on the "Edit" button on a specific quote will take us to the `Quotes#edit` page to update an existing quote.

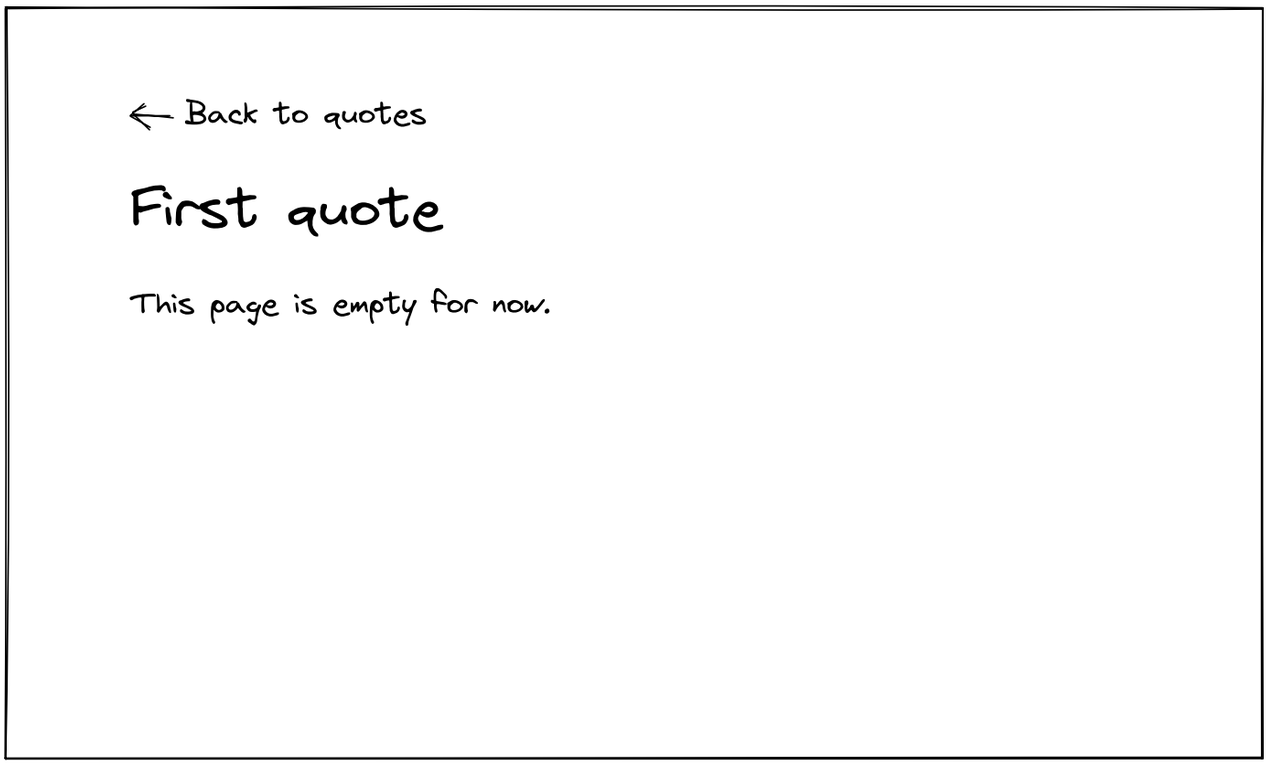
Sketch of the Quotes#edit page

Clicking on the "Update quote" button will redirect us to the `Quotes#index` page, where we can see our updated quote.



Sketch of the Quotes#index page with the updated quote

Last but not least, clicking on the name of a quote will take us to the `Quotes#show` page. It will only contain the quote title for now, but we will add much more features to this page later!



Sketch of the empty Quotes#show page

Ready to get started? First, let's add a few tests to make sure what we are building will work as expected.

Testing our Rails application

Testing is a fundamental part of software development. Without a robust test suite, we will introduce bugs in our application and break features that used to work without even knowing about it. We will make our work more painful, and we want to avoid that.

In this application, we will write Rails system tests to make sure our application always behaves as we want it to. Luckily, those tests are straightforward to write.

Let's first run the generator to create the test file for us:

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

With the help of the sketches above, let's describe what happens in plain English and write some tests at the same time:

```
# 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
```

See how that test reads like plain English? We are very lucky to be working with Ruby on Rails! We need three more tests to show, update, and destroy a quote, but we need some testing data to write them.

This tutorial will use Rails fixtures to create fake data for our tests. Fixtures come with Rails by default and are very easy to use. They consist of a YAML file for each model. Every entry in the YAML fixture files will be loaded into the database before the test suite runs.

Let's first create the fixture file for our quotes:

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

Let's create a few quotes in this file:

```
# test/fixtures/quotes.yml

first:
  name: First quote

second:
  name: Second quote

third:
  name: Third quote
```

We are now ready to add two more tests to our test suite:

```
# 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
```

Now that our tests are ready, we can run them with `bin/rails test:system`. As we can notice, all of them are failing because we are missing routes, a `Quote` model, and a `QuotesController`. Now that our requirements are precise, it's time to start working on the meat of our application.

Creating our Quote model

First, let's create the `Quote` model with a `name` attribute and its associated migration file by running the following command in the console. **As we already generated the fixture file, type "n" for "no" when asked to override it**:

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

All our quotes must have a name to be valid, so we'll add this as a validation in the model:

```
# app/models/quote.rb

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

In the `CreateQuotes` migration, let's add `null: false` as a constraint to our `name` attribute to enforce the validation and ensure we will never store quotes with an empty name in the database even if we made a mistake in the console.

```
# 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
```

We are now ready to run the migration:

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

Adding our routes and controller actions

Now that our model is ready, it's time to develop our CRUD controller. Let's create the file with the help of the rails generator:

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

Let's add the seven routes of the CRUD for our `Quote` resource:

```
# config/routes.rb

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

Now that the routes are all set, we can write the corresponding controller actions:

```
# 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
```

Great! The last thing we need to do to make our tests pass is to create the views.

Note: We are going quite fast here. If you struggle with this controller, you should start by reading the getting started guide from the Ruby on Rails documentation and return to the tutorial when you finish reading it.

Adding our quote views

Note: The views we will add already have a few CSS class names. We will build our design system and create the corresponding CSS files in the next chapter, so, for now, you can just copy the markup. This is for convenience so that we don't have to come back and edit those views in the next chapter.

The markup for the Quotes#index page

The first file we have to create is the `Quotes#index` page `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>
```

We follow the Ruby on Rails conventions for the `Quotes#index` page by rendering each quote in the `@quotes` collection from the partial `app/views/quotes/_quote.html.erb`. This is why we need this second file:

```
<%# 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>
```

The markup for the Quotes#new and Quotes#edit pages

The first files we have to create are `app/views/quotes/new.html.erb` and `app/views/quotes/edit.html.erb`. Note that the only difference between the two pages is the content of the `<h1>` tag.

```
<%# 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>
```

Once again, we will follow Ruby on Rails conventions by rendering the form from the `app/views/quotes/_form.html.erb` file. That way, we can use the same partial for both the `Quotes#new` and the `Quotes#edit` pages.

```
<%# 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 %>
```

The autofocus option is here to focus the corresponding input field when the form appears on the screen, so we don't have to use the mouse and can type directly in it. Notice how the markup for the form is simple? It's because we are going to use the `simple_form` gem. To install it, let's add the gem to our `Gemfile`.

```
# Gemfile

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

With our gem added, it's time to install it:

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

The role of the `simple_form` gem is to make forms easy to work with. It also helps keep the form designs consistent across the application by making sure we always use the same CSS classes. Let's replace the content of the configuration file and break it down together:

```
# 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
```

The *wrappers configuration* lists the input wrappers that we can use in our application. For example, calling `f.input :name` for the `Quote` model with the `:default` wrapper will generate the following 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>
```

Can you see how the CSS classes of the generated HTML match our wrapper's definition? That's what simple form does! It helps us define wrappers for forms that we can reuse in our application very easily, making our forms easy to work with.

The *default configuration* part contains options to configure the submit buttons' default classes, the labels, the way checkboxes and radio buttons are rendered... The most important one here is the `config.default_wrapper = :default`. It means that when we type `f.input :name` without specifying a wrapper, the `:default` wrapper seen above will be used to generate the HTML.

It turns out we will only need the `:default` wrapper in our whole application, so we are done with the `simple_form` gem configuration.

Simple form also helps us define text for labels and placeholders in another configuration file:

```
# 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
```

The configuration above tells simple form that the placeholder for names inputs on quote forms should be "Name of your quote" and that the label for quote names inputs should be "Name".

The last lines under the `:helpers` key can be used without the simple form gem, but we will still add them there as they are related to forms. They define the text of the `submit` buttons on quote forms:

- When the quote is a new record, the submit button will have the text "Create quote"
- When the quote is already persisted in the database, the submit button will have text "Update quote"

The markup for the Quotes#show page

The last view we need is the `Quotes#show` page. For now, it will be almost empty, containing only a title with the name of the quote and a link to go back to the `Quotes#index` page.

```
<%# 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>
```

It seems like we just accomplished our mission. Let's make sure our test passes by running `bin/rails test:system`. They pass!

Note: When launching the system tests, we will see the Google Chrome browser open and perform the tasks we created for our quote system test. We can use the `headless_chrome` driver instead to prevent the Google Chrome window from opening:

```
# 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
```

With this simple change in the configuration, the tests now pass without opening the Google Chrome window every time!

We can also launch our webserver with the `bin/dev` command in the terminal and make sure everything works just fine. Let's not forget to restart our server as we just installed a new gem and modified a configuration file.

Turbo Drive: Form responses must redirect to another location

The app works as expected **unless we submit an empty form**. Even if we have a presence validation on the name of the quote, the error message is not displayed as we could expect when the form submission fails. If we open the console in our dev tools, we will see the cryptic error message "Form responses must redirect to another location".

This is a "breaking change" since Rails 7 because of *Turbo Drive*. We will discuss this topic in-depth in Chapter 3 dedicated to *Turbo Drive*. If you ever encounter this issue, the way to fix it is to add `status: :unprocessable_entity` to the `QuotesController#create` and `QuotesController#update` actions when the form is submitted with errors:

```
# 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
```

With the response status code added, our CRUD is now working perfectly, but the user interface is really ugly. In the next chapter, we will fix that where we will create a small design system for our quote editor (the one that I use on this website!).

Seeding our application with development data

When we launched our server for the first time, we didn't have any quotes on our page. Without development data, every time we want to edit or delete a quote, we must create it first.

While this is fine for a small application like this one, it becomes annoying for real-world applications. Imagine if we needed to create all the data manually every time we want to add a new feature. This is why most Rails applications have a script in `db/seeds.rb` to populate the development database with fake data to help set up realistic data for development.

In our tutorial, however, we already have fixtures for test data:

```
# test/fixtures/quotes.yml

first:
  name: First quote

second:
  name: Second quote

third:
  name: Third quote
```

We will reuse the data from the fixtures to create our development data. It will have two advantages:

- We won't have to do the work twice in both `db/seeds.rb` and the fixtures files
- We will keep test data and development data in sync

If you like using fixtures for your tests, you may know that instead of running `bin/rails db:seed` you can run `bin/rails db:fixtures:load` to create development data from your fixtures files. Let's tell Rails that the two commands are equivalent in the `db/seeds.rb` file:

```
# db/seeds.rb

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

Running the `bin/rails db:seed` command is now equivalent to removing all the quotes and loading fixtures as development data. Every time we need to reset a clean development data, we can run the `bin/rails db:seed` command:

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

Now that we set up everything, we might notice that the design is well... Not designed at all! In the next chapter, we will write some CSS to create the components we need for our quote editor. See you there!

← previous next →