Hotrails

← Back to the list of chapters

Turbo Streams and security

Published on February 10, 2022

In this chapter, we will learn how to use Turbo Streams securely and avoid sending broadcastings to the wrong users.

Understanding Turbo Streams and security

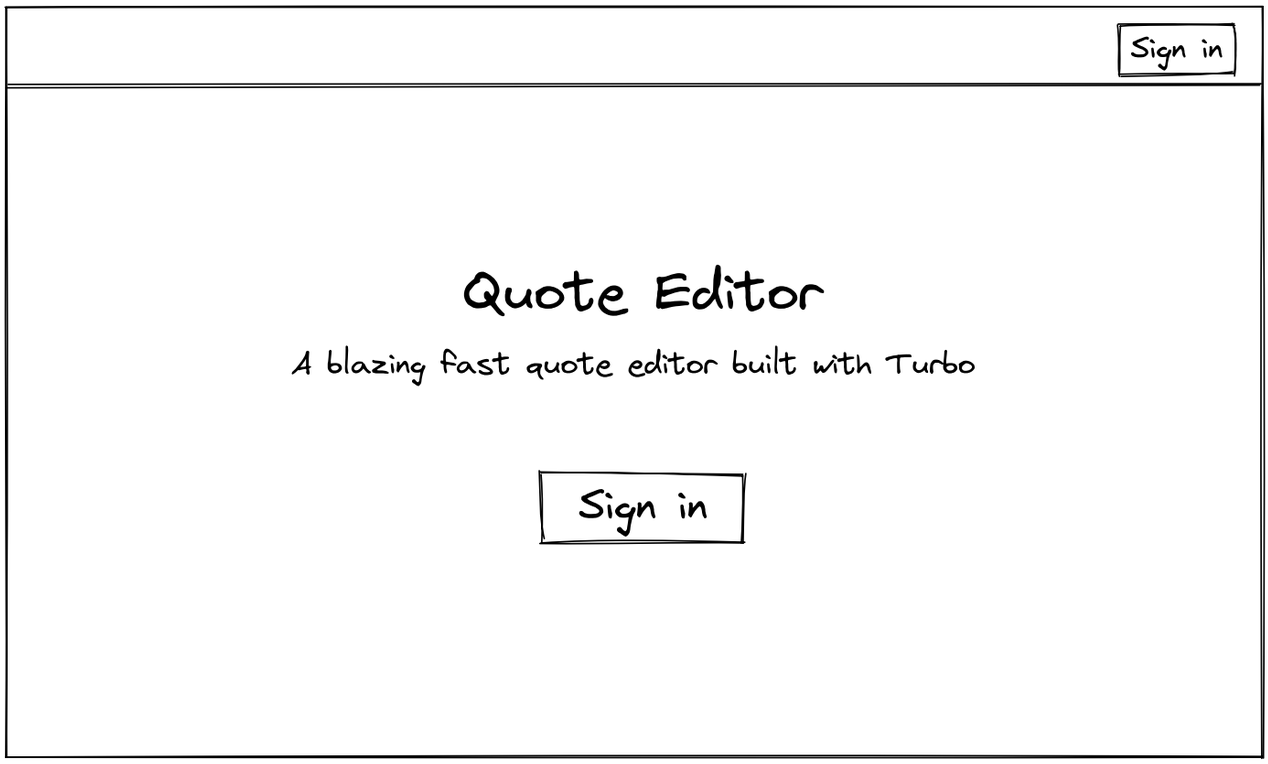
Before using Turbo Streams in production, **we must understand how security works. It would be terrible to broadcast HTML containing sensitive data to a user that should not receive it**.

Let's imagine that our quote editor was a more complex application where quotes belong to companies and companies have many users. In that case, we should not broadcast quotes to the `Quotes#index` page of a user that does not belong to our company. **That would be a significant security flaw**.

We will add **users** with the Devise gem and **companies** to our application to simulate this real-world scenario and understand Turbo Streams security. We will then experiment in the browser to show some of the security issues that might arise with Turbo Streams if we are not careful enough.

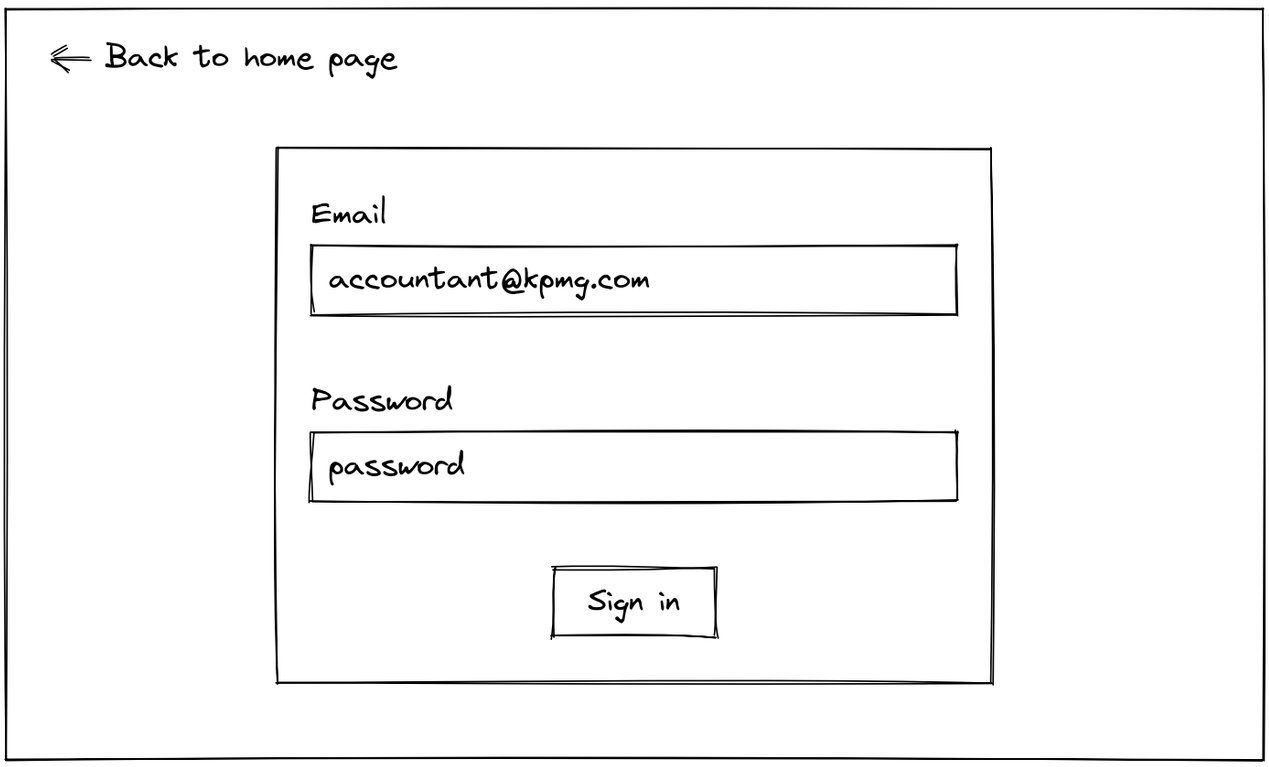
What we will build

Let's sketch what our application will look like at the end of this chapter. We will add a home page with links to the login page when the user is not signed in:



Sketch of the home page when the user is not signed in

Our users will be able to sign in by entering their email and password:

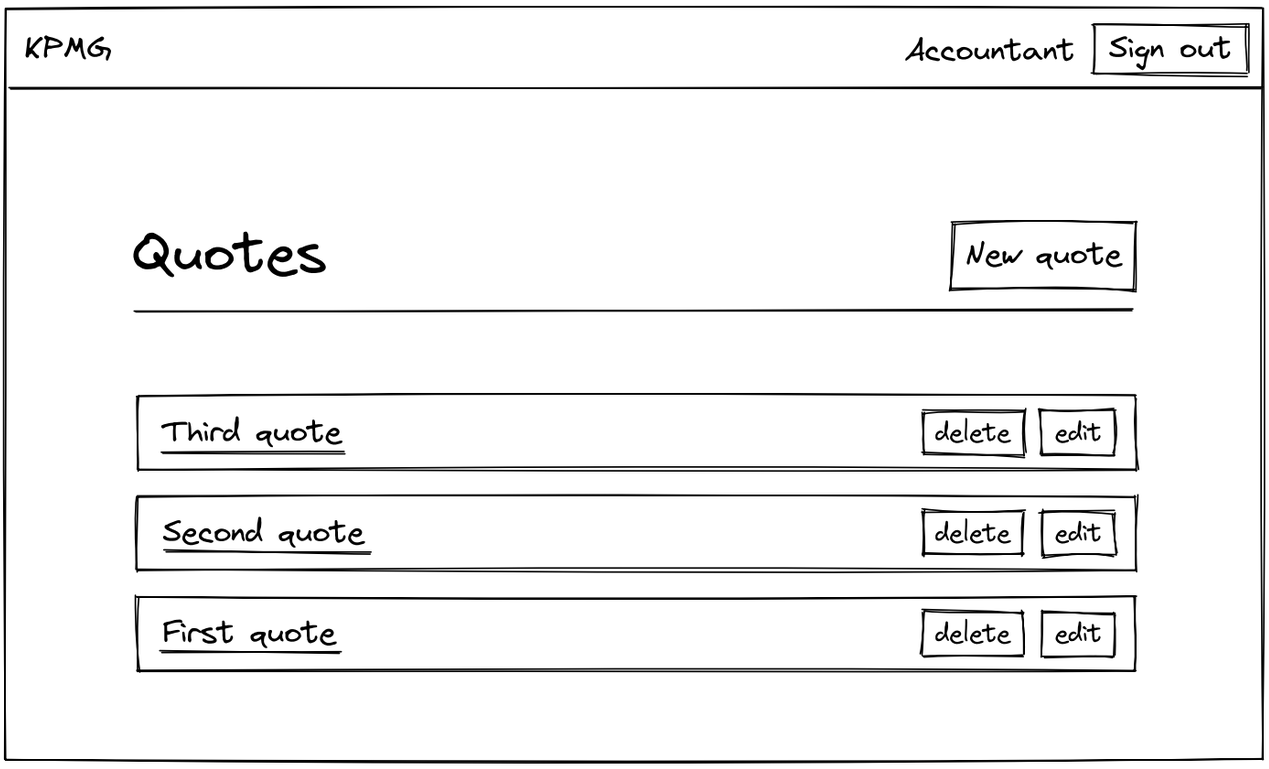


Sketch of the login page

Our users will be redirected to the home page when they sign in. In the navbar, we will display the name of the company and user's *name* based on the email address. They will be able to navigate to our quote editor by clicking on the "View quotes" button:

Sketch of the home page when the user is signed in

When clicking on the "View quotes" button, users will navigate to the `Quotes#index` page that will now have a navbar:



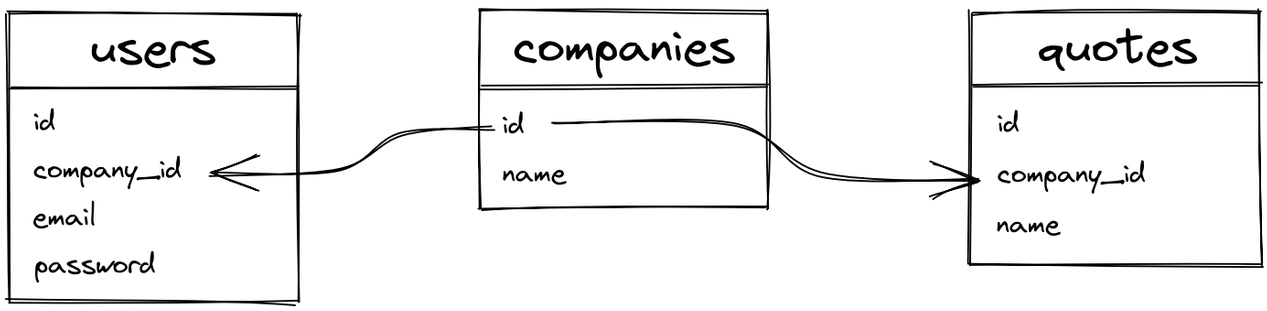
Sketch of the Quotes#index page with the navbar

With all of these new additions, our application will start looking like a real one, and we will be able to do some experiments in the browser where we will find a security issue.

The `Quote`, `Company`, and `User` models will be related to one another by the following associations:

- A **user** belongs to a **company**
- A **quote** belongs to a **company**
- A **company** has many **users**
- A **company** has many **quotes**

The database schema we will implement is illustrated in the following sketch:



Sketch of the desired database schema

We will seed the data we need with the `rails db:seed` command to simulate our real-world scenario. In the fixtures, we will need two companies and three users:

- The first company (**KPMG**) will have two users: an **accountant** and a **manager**.
- The second company (**PwC**) will have only one user: an ***eavesdropper*** who should never access to **KPMG**'s quotes.

Let's create the `Company` and the `User` models, and add the required associations to match our database schema.

Adding companies to our Rails application

Let's create the `Company` model:

```
rails generate model Company name
```

Let's edit the migration because we want our companies always to have a name. It is good to enforce this at the database level by adding `null: false` as a database constraint. It will prevent any companies from having an empty name if, for some reason, validations are skipped:

```
# db/migrate/XXXXXXXXXXXXXX_create_companies.rb
```

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

      t.timestamps
    end
  end
end
```

We can now run the migration:

```
rails db:migrate
```

Let's not forget to add the presence validation for the name in the `Company` model:

```
# app/models/company.rb

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

We will only need the two companies we talked about in introduction: **KPMG** and **PwC**. Let's add them to our `companies.yml` fixture file:

```
# test/fixtures/companies.yml

kpmg:
  name: KPMG

pwc:
  name: PwC
```

We only need those two companies to make our real-world example work. To talk about Turbo Streams and security, we don't need to create the full CRUD in the `CompaniesController` and the associated views. Let's continue by adding users to our application.

Adding users to our application with Devise

We will use the very popular and widely used Devise gem to add users to our application and authenticate them.

Let's add it into our `Gemfile`:

```
# Gemfile

gem "devise", "~> 4.8.1"
```

Let's now install the gem:

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

We can now generate the `User` model with the Devise generators:

```
bin/rails generate devise User
bin/rails db:migrate
```

Now that we have our `User` model in place, we need a view to sign in to match the sketches we described in the introduction. We won't allow users to register because we would have to code the logic to attach new users to their companies. We don't need all this work to talk about Turbo Streams and security, so let's keep things simple.

We just need users in our seeds and a way to log in. Therefore, we will disable all the Devise features for our `User` model except two of them:

- The feature to sign in users (`:database_authenticatable`)
- The feature to validate the email and password using Devise built-in validations (`:validatable`)

Here is how our `User` model should look like:

```
# app/models/user.rb

class User < ApplicationRecord
  devise :database_authenticatable, :validatable
end
```

Last but not least, let's create the fixtures we talked about in the introduction:

```
# test/fixtures/users.yml

accountant:
  email: accountant@kpmg.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>

manager:
  email: manager@kpmg.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>

eavesdropper:
  email: eavesdropper@pwc.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>
```

Devise stores the `encrypted_password` field in the `users` database table for security reasons. If we want our fixtures to have the string `"password"` as a password, we need to use the same method the Devise gem would use to encrypt the password in our fixtures. This is why we use the `Devise::Encryptor.digest` method here.

With our `User` model complete, **we have to add the associations between users, companies, and quotes**. That's what we will do in the next section.

Note: When logging in with users, you might encounter a redirection bug when submitting an invalid form. This is because **the Devise gem does not support Turbo yet (version 4.8.1)**. The easiest way to prevent this bug is to **disable Turbo on Devise forms** by setting the `data-turbo` attribute to `false` on Devise forms, as we learned in the Turbo Drive chapter.

We won't do it in our Tutorial, but if we pushed our app to production, we would have to do it before real users try our app.

Users, companies and, quotes associations

We don't have any associations between our `User`, `Company`, and `Quote` models. Let's generate two migrations to be able to add those associations.

```
bin/rails generate migration add_company_reference_to_quotes company:ref
bin/rails generate migration add_company_reference_to_users  company:ref
```

The first migration will add the `company_id` foreign key to quotes, and the second will add the `company_id` foreign key to users. Thanks to the Rails generators, those two migrations are ready for a migration:

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

Note: The migrations will fail if we have some users or quotes in our database because the two migration files specify `null: false` as a constraint for the `company_id` foreign key on users and quotes. The quotes or users in our database currently have a blank `company_id` which clashes with the new constraint.

If our project were a real application already in production, we would have to first populate the `company_id` field for all users and quotes before adding the `null: false` constraint. As our application is not yet in production, we can simply drop the database, re-create it and rerun our migration:

```
bin/rails db:drop db:create db:migrate
```

Our migration now runs successfully!

Now that our migrations pass and our database schema is complete, let's add the associations on the `User`, `Company`, and `Quote` models:

```
# app/models/user.rb

class User < ApplicationRecord
  devise :database_authenticatable, :validatable

  belongs_to :company
end
```

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

class Company < ApplicationRecord
  has_many :users, dependent: :destroy
  has_many :quotes, dependent: :destroy

  validates :name, presence: true
end
```

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

class Quote < ApplicationRecord
  belongs_to :company

  # All the previous code
end
```

Let's update our fixtures accordingly. For our users' fixtures, we mentioned in the introduction that our **accountant** and our **manager** should belong to **KPMG**. On the other hand, our **eavesdropper** should belong to **PwC**. This is very easy to do with fixtures:

```yaml
# test/fixtures/users.yml

accountant:
  company: kpmg
  email: accountant@kpmg.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>

manager:
  company: kpmg
  email: manager@kpmg.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>

eavesdropper:
  company: pwc
  email: eavesdropper@pwc.com
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>
```

Let's also update our quotes fixtures as they belong to a company. All the quotes we used until now will belong to **KPMG**, and **PwC** won't have any quotes by default:

```
# test/fixtures/quotes.yml

first:
  company: kpmg
  name: First quote

second:
  company: kpmg
  name: Second quote

third:
  company: kpmg
  name: Third quote
```

Our fixtures are now ready. Let's seed the database by running `rails db:seed` in the console. Everything is set up!

Adding a home page to our application

Now that we have users in the application, we must help them easily sign in. As described in the sketches from the beginning of the chapter:

1. Users must be authenticated to access the quote editor, and they should only see quotes that belong to their company.
2. Users must be able to navigate to the sign-in form from the home page even when they are not authenticated.

To solve the first bullet point above, we need to ensure our users are authenticated everywhere on the whole application. Let's enforce that in the `ApplicationController` thanks to the Devise method `authenticate_user!`:

```
# app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  before_action :authenticate_user!
end
```

To solve the second bullet point, we need unauthenticated users to be able to access the login form; otherwise, they couldn't sign in. Let's add an exception to our callback:

```
# app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  before_action :authenticate_user!, unless: :devise_controller?
end
```

We also need a home page from which our users can navigate to the sign-in form. Let's create a `PagesController` with a `home` action as a root path. This controller is public, so we will skip the need to be authenticated:

```
# app/controllers/pages_controller.rb

class PagesController < ApplicationController
  skip_before_action :authenticate_user!

  def home
  end
end
```

Let's now make the `home` action the root path of our application:

```
# config/routes.rb

Rails.application.routes.draw do
  root to: "pages#home"

  # All the other routes
end
```

Finally, let's add the corresponding view and make it look like our sketches:

```
<%# app/views/pages/home.html.erb %>

<main class="container">
  <h1>Quote editor</h1>
  <p>A blazing fast quote editor built with Hotwire</p>

  <% if user_signed_in? %>
    <%= link_to "View quotes", quotes_path, class: "btn btn--dark" %>
  <% else %>
    <%= link_to "Sign in", new_user_session_path, class: "btn btn--dark"
```

```
  <% end %>
</main>
```

We won't style our landing page more than that, but we will spend more time on the navbar as it will be visible everywhere in our application.

Let's first add the markup for our navbar. We will first use *placeholders* for the company's name and the current user's name, but we will change that soon. For now, let's focus on the HTML:

```
<%# app/views/layouts/_navbar.html.erb %>

<header class="navbar">
  <% if user_signed_in? %>
    <div class="navbar__brand">
      Company name
    </div>
    <div class="navbar__name">
      Current user name
    </div>
    <%= button_to "Sign out",
                  destroy_user_session_path,
                  method: :delete,
                  class: "btn btn--dark" %>
  <% else %>
    <%= link_to "Sign in",
                new_user_session_path,
                class: "btn btn--dark navbar__right" %>
  <% end %>
</header>
```

To render our navbar on every page of our application, we can render it directly from the application's layout:

```
<%# app/views/layouts/application.html.erb %>

<!DOCTYPE html>
<html>
  <!-- All the <head> content -->

  <body>
    <%= render "layouts/navbar" %>
```

```
    <%= yield %>
  </body>
</html>
```

Now that our navbar is displayed on the whole application, we will write a little bit of CSS to style it:

```
// app/assets/stylesheets/components/_navbar.scss

.navbar {
  display: flex;
  align-items: center;
  box-shadow: var(--shadow-large);
  padding: var(--space-xs) var(--space-m);
  margin-bottom: var(--space-xxl);
  background-color: (var(--color-white));

  &__brand {
    font-weight: bold;
    font-size: var(--font-size-xl);
    color: var(--color-text-header);
  }

  &__name {
    font-weight: bold;
    margin-left: auto;
    margin-right: var(--space-s);
    color: var(--color-text-header);
  }

  &__right {
    margin-left: auto;
  }
}
```

Let's not forget to import this CSS file into our manifest file:

```
// app/assets/stylesheets/application.sass.scss

// All the previous imports
@import "components/navbar";
```

Everything should start looking exactly like the sketches in the first section of this chapter, except we won't style the sign-in form. Let's test everything is wired up correctly in the browser. **Before we test, let's make sure our seeds are ready** with the `bin/rails db:seed` command.

Let's sign in with our accountant user. To do this, let's navigate to the home page and click on the "Sign in" button. Let's enter the email (`accountant@kpmg.com`) and the password (`password`) for the accountant fixture n the sign in page. Our navbar looks nice, but we still need to change the company's name and current user's name dynamically.

Let's start with the easiest one: the name of the `current_user` . We will use it to see which user fixture is logged in to make it easier for us in development. Let's add a `#name` method to our `User` model to guess the user's name from the email address:

```
# app/models/user.rb

class User < ApplicationRecord
  devise :database_authenticatable, :validatable

  belongs_to :company

  def name
    email.split("@").first.capitalize
  end
end
```

We can also add a test to our method to ensure it behaves as we expect:

```
# test/models/user_test.rb

require "test_helper"

class UserTest < ActiveSupport::TestCase
  test "name" do
    assert_equal "Accountant", users(:accountant).name
  end
end
```

Now that we are sure our method displays the correct result, we can update the HTML in our navbar:

```
<%# app/views/layouts/_navbar.html.erb %>

<header class="navbar">
  <% if user_signed_in? %>
    <div class="navbar__brand">
      Company name
    </div>
    <div class="navbar__name">
      <%= current_user.name %>
    </div>
    <%= button_to "Sign out",
                  destroy_user_session_path,
                  method: :delete,
                  class: "btn btn--dark" %>
  <% else %>
    <%= link_to "Sign in",
                new_user_session_path,
                class: "btn btn--dark navbar__right" %>
  <% end %>
</header>
```

We also want to add the company name to the navbar. Like we have a `current_user`, it would be nice to have a `current_company` as our users always belong to a company.

Let's add this logic into the `ApplicationController` as we will use the `current_company` method later in our controllers and views. To use the `current_company` method in our views, we need to turn it into a helper. We can do that thanks to the `helper_method` method:

```
# app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  before_action :authenticate_user!, unless: :devise_controller?

  private

  def current_company
    @current_company ||= current_user.company if user_signed_in?
  end
```

```
  helper_method :current_company
end
```

Now that we have our `current_company` helper, we can use it in our views and especially in our navbar:

```
<%# app/views/layouts/_navbar.html.erb %>

<header class="navbar">
  <% if user_signed_in? %>
    <div class="navbar__brand">
      <%= current_company.name %>
    </div>
    <div class="navbar__name">
      <%= current_user.name %>
    </div>
    <%= button_to "Sign out",
                  destroy_user_session_path,
                  method: :delete,
                  class: "btn btn--dark" %>
  <% else %>
    <%= link_to "Sign in",
                new_user_session_path,
                class: "btn btn--dark navbar__right" %>
  <% end %>
</header>
```

That was a lot of setup code, but we are almost there. We only need to fix our tests because we required users to be signed in on the quote editor and added some associations. After that, we will be ready to discuss Turbo Streams and security by doing experimenting with the browser.

Fixing our tests

Our system tests are currently broken! Let's run them with the `bin/rails test:system` command.

The first error we might notice is that our users now need to sign in before manipulating quotes. To sign in users in system tests, we will rely on helpers from the `Warden` gem to avoid coding them ourselves. Devise is built on top of

`Warden` and the `Warden::Test::Helpers` module contains helpers that will help us login users in our tests thanks to the `login_as` method.

To use them, let's first include `Warden::Test::Helpers` in our `ApplicationSystemTestCase` class.

```
# test/application_system_test_case.rb

require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  include Warden::Test::Helpers

  driven_by :selenium, using: :headless_chrome, screen_size: [1400, 1400
end
```

We can now use the helpers from the `Warden::Test::Helpers` module in our system tests as all of our system tests classes inherit from `ApplicationSystemTestCase`. What we need to do, is to log in our accountant user before each test run in the quote system test. To do this, we will use the `login_as` helper from the `Warden::Test::Helpers` in the setup block:

```
# test/system/quotes_test.rb

require "application_system_test_case"

class QuotesTest < ApplicationSystemTestCase
  setup do
    login_as users(:accountant)
    @quote = Quote.ordered.first
  end

  # All the previous code
end
```

Let's rerun the system tests with the `bin/rails test:system` command. Some tests are still failing because we require quotes to be associated with a company. Let's update the `QuotesController` to use the association with the `Company` model.

The `QuotesController#index` method should only show the quotes that belong to the current user's company. Let's use the associations for this:

```
# app/controllers/quotes_controller.rb

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

Also, when creating the quote, we need to make sure the quote is associated with the current user's company:

```
# app/controllers/quotes_controller.rb

def create
  # Only this first line changes to make sure the association is created
  @quote = current_company.quotes.build(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
  end
end
```

For all the other actions, we must ensure the quote we manipulate is scoped to the `current_company` to prevent users from manipulating quotes that don't belong to their company, thus avoiding security issues:

```
class QuotesController < ApplicationController
  before_action :set_quote, only: [:show, :edit, :update, :destroy]
  # All the previous code

  private

  def set_quote
    # We must use current_company.quotes here instead of Quote
    # for security reasons
    @quote = current_company.quotes.find(params[:id])
```

```
  end

  # All the previous code
end
```

Let's run our system tests one more time with the `bin/rails test:system` command. All should be green. Let's run *all* of our tests with the `bin/rails test:all` command. They all pass! We are now ready to discuss Turbo Streams and security.

Security and Turbo Streams

This was a very lengthy setup, but we are ready to discuss how security works with Turbo Streams.

Let's start by showing a big problem we have in our application. To do this, let's open two browser windows **side by side to be able to see real-time updates**:

- One browser window in default mode
- One browser window in private navigation

In the default mode window, let's sign in our **accountant fixture** defined in the `users.yml` fixtures file (the email is `accountant@kpmg.com`, and the password is `password`)!

In the private navigation window, let's sign in with our **eavesdropper fixture** (the email is `eavesdropper@pwc.com` and the password also is `password`).

Now let's make both these users navigate to the `Quotes#index` page. With the **accountant**'s account, let's create a quote named "Secret quote". The "Secret quote" appears in the private window where the **eavesdropper** is logged in.

There is a critical security issue here: our quote created by the **accountant** was broadcasted to the **eavesdropper** user. Those two users **belong to different companies** and **should never gain access to the other company's quotes**.

If we refresh the private navigation page with our **eavesdropper** account, the "Secret quote" disappears. This is because the HTML containing the "Secret

quote" was broadcasted to the **eavesdropper** user even if we correctly scoped the list of quotes to the `current_company` in the `QuotesController#index` action:

```
# app/controllers/quotes_controller.rb

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

This is a Turbo Streams issue. Before using Turbo Streams in production, we first need to understand how security works at a high level. Let's analyze why we have this security breach and how to fix it.

Turbo Stream Security in depth

It is essential to understand how security works before using `turbo-rails` in production. **Security is a complex topic, but the solution is straightforward here**.

Let's start our journey by noticing something. Let's open our dev tools and inspect the DOM with our **accountant**'s session and our **eavesdropper**'s session (the default navigation window and the private one). In the `Quotes#index` page on both browser windows, we should find a `<turbo-cable-stream-source>` tag. It was generated by the `turbo_stream_from` helper we called in the `Quotes#index` page:

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

<%# This line generate the <turbo-cable-stream-source> tag %>
<%= turbo_stream_from "quotes" %>

<%# All previous content %>
```

Let's copy the `turbo-cable-stream-source` from both the accountant's session and the eavesdropper's session.

```
<!-- Accountant's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
```

```
  signed-stream-name="InF1b3RlcyI=--eba9a5055d229db025dd2ed20d069d87c36a
>
</turbo-cable-stream-source>

<!-- Eavesdropper's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="InF1b3RlcyI=--eba9a5055d229db025dd2ed20d069d87c36a
>
</turbo-cable-stream-source>
```

Notice how **both signed stream names are the same**? We begin to understand why both users receive the broadcastings for quotes creation, updates, and deletion.

Note: The attribute name for the stream is `signed-stream-name` instead of simply `stream-name`. The `turbo_stream_from` helper automatically signs the stream name to prevent users from tampering with its value and gaining access to private broadcastings. The turbo-rails gem takes care of this security concern for us!

In the sketches, I use a simplified version `stream-name` for our mental model. In reality, the `stream-name` is signed automatically, and we don't have to think about it.

Note: Your `signed-stream-name` won't be the same as mine because it won't be signed with the same private key. Rails generate a different private key for every new Ruby on Rails application. What is important here is that the `signed-stream-name` attributes are the same for both the **accountant** and the **eavesdropper**.

Both users have subscribed to the `Turbo::StreamsChannel` thanks to the `channel` attribute. All communications between the publisher (the server) and the subscriber (the client) from Turbo Streams will go through the `Turbo::StreamsChannel`.

However, both users have the same `signed-stream-name`. In `ActionCable`, the role of a stream is to route broadcastings to subscribers. Therefore, if we have the same `signed-stream-name` on both the **accountant**'s and **eavesdropper**'s `Quotes#index` page, they will both receive the same broadcastings. This is why when a quote is created, the corresponding HTML is broadcasted to both the **accountant** and the **eavesdropper**!

Let's sketch what happens. When the **accountant** creates the "Secret quote", it is broadcasted to the stream names "quotes". Both the **accountant** and the **eavesdropper** are receiving the broadcastings as they are streaming from the "quotes" stream.

The accountant is streaming from the quotes stream

The eavesdropper is also streaming from the quotes stream

To solve our security issue, those `signed-stream-name` attributes must be ***different***.

Now that we understand the problem better, we will see how to solve it in the next section.

Fixing our Turbo Streams security issue

In the previous chapter, we instructed the quote model to broadcast creations, updates, and deletions to users:

```
# app/models/quote.rb

class Quote < ApplicationRecord
  # All the previous code

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

The part that interests us here is the lambda passed as the first argument to the `broadcasts_to` method:

```
->(quote) { "quotes" }
```

What this does under the hood is that the `Turbo::StreamsChannel` will broadcast quotes creations, updates, and deletions through the `"quotes"` stream as this lambda will always return the string `"quotes"`. The broadcasting is received in the `Quotes#index` page because users are subscribed to stream thanks to the following line:

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

<%# This line generate the <turbo-cable-stream-source> tag %>
<%= turbo_stream_from "quotes" %>

<%# All previous content %>
```

We want to have **the same** `signed-stream-name` for the **accountant** and the **manager** and **a different one** for the **eavesdropper**. To do this, we have to change the stream name where the quotes' HTML will be broadcasted. Doing this with `turbo-rails` is very simple:

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

  broadcasts_to ->(quote) { [quote.company, "quotes"] }, inserts_by: :pr
end
```

Under the hood, the signed stream name is generated from the array returned by the lambda that is the first argument of the `broadcasts_to` method. The rules for secure broadcastings are the following:

1. **Users who share broadcastings should have the lambda return an array with the same values.**
2. **Users who shouldn't share broadcastings should have the lambda return an array with different values.**

In our example, the quote's company is the *same* for the **accountant** and the **manager** fixture users. For both of them, the lambda returns an array with the

same values, so they can share quotes creations, updates, and deletions broadcastings.

The quote's company is *different* for the **eavesdropper**, so he will not be able to receive the broadcastings.

To make our feature work again, we need to update the `turbo_stream_from` in the `Quotes#index` page as we just changed the stream name in our `Quotes#index` view to match the values inside the lambda:

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

<%= turbo_stream_from current_company, "quotes" %>
```

Let's experiment again in the browser. Let's open the `Quotes#index` page in a default browser window with our **accountant**'s session and the `Quotes#index` page in a private navigation window with our **eavesdropper**'s session. Let's create a new quote with the name of "Secret quote" with our **accountant**'s session. This time, it was not prepended to the **eavesdropper**'s list of quotes!

By inspecting the DOM, we can see that the `signed-stream-name` values are no longer the same on the **accountant**'s and the **eavesdropper**'s `Quote#index` page.

```
<!-- Accountant's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="IloybGtPaTh2YUc5MGQybHlaUzFqYjNWeWMyVXZRMjl0Y0dGdW
>
</turbo-cable-stream-source>

<!-- Eavesdropper's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="IloybGtPaTh2YUc5MGQybHlaUzFqYjNWeWMyVXZRMjl0Y0dGdW
>
</turbo-cable-stream-source>
```

Let's now do the test with the **accountant** and the **manager** accounts. Let's log the **eavesdropper** out in the private window and log the **manager** in. With both users on the `Quotes#index` page, let's create a "Shared quote" quote with the

accountant's account. The created quote should be prepended in real-time to the list of quotes in the **manager**'s private window. Everything now works as expected!

By inspecting the DOM, we can see that the `signed-stream-name` values are the same on the **accountant**'s and the **manager**'s `Quote#index` page.

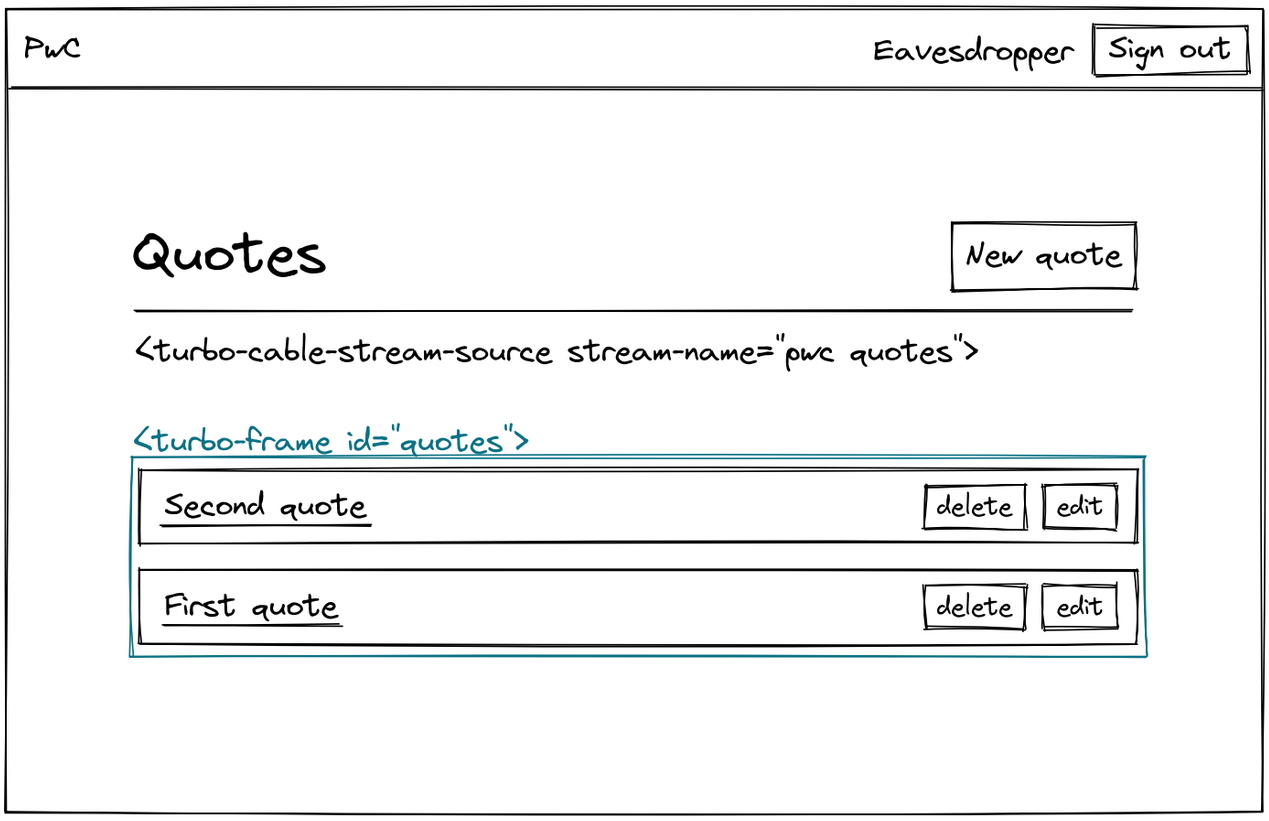
```
<!-- Accountant's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="IloybGtPaTh2YUc5MGQybHlaUzFqYjNWeWMyVXZRMjl0Y0dGdW
>
</turbo-cable-stream-source>

<!-- Manager's session -->
<turbo-cable-stream-source
  channel="Turbo::StreamsChannel"
  signed-stream-name="IloybGtPaTh2YUc5MGQybHlaUzFqYjNWeWMyVXZRMjl0Y0dGdW
>
</turbo-cable-stream-source>
```

Let's sketch what happens here. The **accountant** and the **manager** both receive the broadcasting as they share the same stream name:

The accountant receives the broadcasting

The manager receives the broadcasting

The **eavesdropper** does not receive the broadcasting as the `<turbo-cable-stream-source>` element does not have the same `stream-name` attribute:



The eavesdropper does not receive the broadcasting

When using Turbo Streams in production, we must ensure that the `broadcasts_to` method in the model and the `turbo_stream_from` method in the view are appropriately configured to avoid security issues.

Wrap up

Turbo Streams are a fantastic tool, but we need to be careful who we broadcast information to if we don't want to create significant security issues.

In this chapter, we saw that this should be configured in the model callbacks that trigger the broadcastings. The lambda in the first argument of the `broadcasts_to` method should return the **same value** for users who share broadcastings and a **different one** for users who should not share the broadcastings.

← previous next →

If you liked this article and want to keep up with Ruby on Rails and Hotwire, you can subscribe to my newsletter (no spam, no tracking, unsubscribe any time)!

Subscribe to the newsletter

Github Twitter Newsletter

Made with remotely