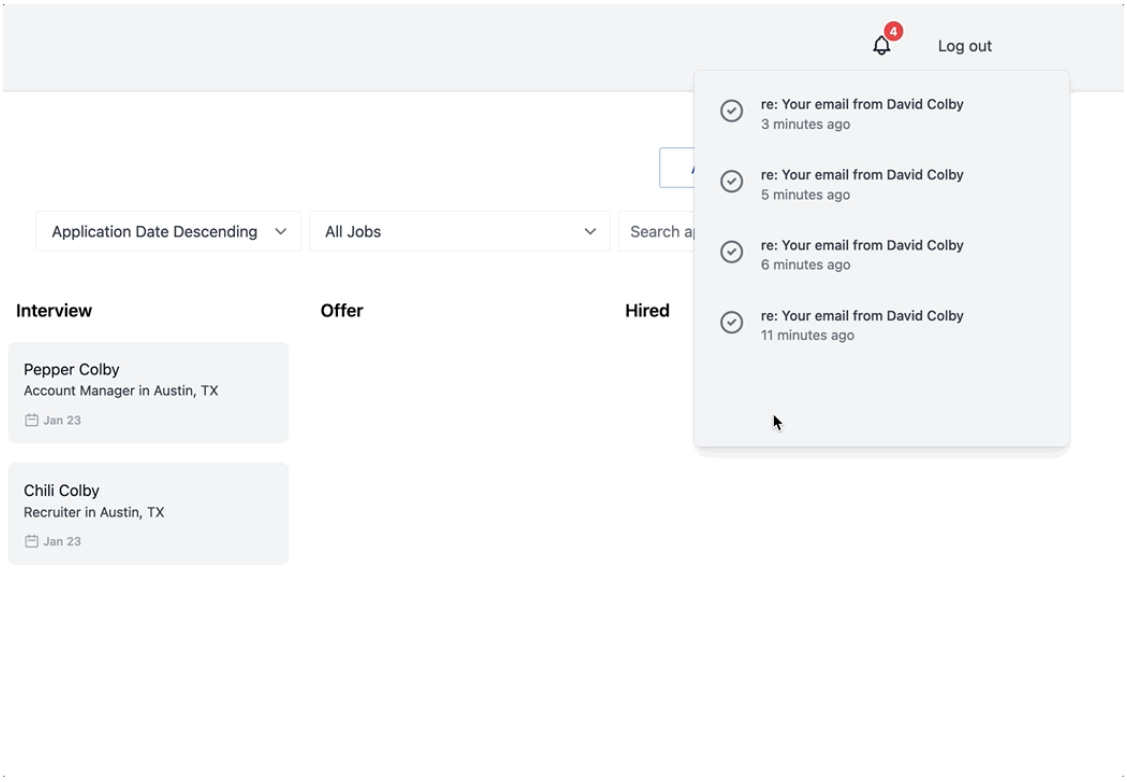
Building a notification system

In the last chapter we added a two-way email system, allowing applicants to reply to emails sent to them by our users. The trouble with this feature is that users have no way of knowing when an applicant has sent them a reply without opening every applicant's show page.

Opening every applicant's record multiple times a day waiting to see if they have responded is not realistic. We can make this feature much more useful by adding in-app notifications each time a user receives a new inbound email.

To do this, we will build a notification system inspired by the wonderful **Noticed gem** and powered by Turbo Streams and StimulusReflex. When we are finished, logged in users will have a new notification menu item in the main nav that automatically updates when new notifications are received.



Before diving in, let's talk about why we are building our own notification system instead of just using Noticed. Noticed is the gold standard for building notification systems in Rails applications, in my opinion. It provides everything you will need right out of the box, is extremely powerful, extensible, and easy to work with.

But — Noticed has a ton of features that we will not be using in our application. The extra complexity can be a little overwhelming if you do not need it. Building our own system also helps us learn how Noticed is structured, which makes it much easier to work with when it is time to build a commercial-grade application.

Head into this chapter knowing that we are building something heavily-inspired by Noticed. After this chapter you should be well-equipped to move to Noticed in your own applications and I encourage you to use Noticed instead of rolling your own notification system outside of learning projects.

Create notification model and relationships

At the end of this chapter we will send inbound email notifications to users. Later in this book we will add notifications for other events. Since we know that we plan to have many types of notifications, we will build our `Notification` model to support **single table inheritance**.

This structure allows us to define the basic behavior of all notification types in the `Notification` class while using subclasses to add unique behavior to each type of notification. We will walk through this together.

Let's start with generating the `Notification` model. From your terminal:

```
rails g model Notification user:references read_at:datetime
params:jsonb type:string
rails db:migrate
```

Notifications `belong_to` a user. The `type` column is the default column name for inheritance in Rails. Finally, the `params` column is a jsonb column that we will use to store the data each notification needs to render itself.

This params **approach** is what you will find used in the Noticed gem and it helps us avoid complicating the data model or running expensive queries to render notifications, which we will see as we progress through this chapter.

Next, update the `User` model to add the `has_many` relationship with notifications:

app/models/user.rb

```
8 | has_many :notifications, dependent: :destroy
```

And then update the `Notification` model to add an `unread` scope and to **serialize** the params column.

app/models/notification.rb

```
1 | class Notification < ApplicationRecord
2 |   include Rails.application.routes.url_helpers
3 |
4 |   belongs_to :user
5 |
6 |   scope :unread, -> { where(read_at: nil) }
7 |
8 |   serialize :params
9 | end
```

We are also including Rails `url_helpers` in the base class because each subclass will need to access path helpers (like `applicant_path`). Now that we have the base `Notification` model in place, next we will build the notification class for inbound email notifications.

Each type of notification will be a new class that inherits from the `Notification` class.

Start from your terminal:

```
mkdir app/notifications
touch app/notifications/inbound_email_notification.rb
```

Here we created a new `notifications` directory where we will create each new notification type, starting with `InboundEmailNotification`. Before moving on, restart your server to avoid potential Zeitwerk problems with picking up the new directory.

After you restart the Rails server, fill in `app/notifications/inbound_email_notification.rb` next:

app/notifications/inbound_email_notification.rb

```
class InboundEmailNotification < Notification
  def message
    "#{params[:email].subject} from #
{params[:applicant].name}"
  end

  def url
    applicant_email_path(params[:applicant],
params[:email])
  end
end
```

Notice `InboundEmailNotification` inherits from `Notification`, giving it all the same behavior as the base class. Each time we add a new notification, we will add a new class and create the `message` and `url` methods for the subclass.

Here the `url` method uses `applicant_email_path`, which we can safely use in the model because of the `include Rails.application.routes.url_helpers` added to the `Notification` class.

These presentational methods will be used to display the notification in the UI and will typically rely on the serialized `params` column. Each time we create a new `InboundEmailNotification` we save the email's content and applicant information in the `params` for reference in the `message` and `url` methods.

Let's see how to use this new class by updating the `Email` model:

app/models/email.rb

```
15  after_create_commit :create_notification, if: :inbound?
16
17  def create_notification
18    InboundEmailNotification.create(
19      user: user,
20      params: {
21        applicant: applicant,
22        email: self
23      }
24    )
25  end
```

This new `after_create_commit` callback only runs for inbound emails since we do not need to notify users about their own outbound emails. The `create_notification` method creates a new Notification with a type of `InboundEmailNotification`, passing in the params needed to render this notification later.

If you are new to using single table inheritance in Rails, the `InboundEmailNotification.create` call is equivalent to `Notification.create(type: 'InboundEmailNotification')`.

At this point creating notifications for new inbound emails is working, but we are not displaying them to users. Let's tackle that next by adding a notification menu to the authenticated navigation bar.

Add notifications to the UI

Notifications will be displayed in a dropdown menu accessed from the navigation bar and loaded in via a notifications Turbo Frame, similar to how we are loading emails on the applicants show page.

First up, generate a new controller and create the views we need from your terminal:

```
rails g controller Notifications
touch app/views/notifications/index.html.erb
touch app/views/notifications/_notification.html.erb
```

And then fill in the notifications index view:

app/views/notifications/index.html.erb

```
<%= turbo_frame_tag "notifications" do %>
  <ul class="p-2" role="list" aria-
orientation="vertical" aria-labelledby="notifications-
list">
    <%= render @notifications %>
  </ul>
<% end %>
```

Here we are using Rails collection rendering to render all `**@notifications**`. For collection rendering to work with the multiple class names that come with using single table inheritance we need to update the `**Notification**` model to override the default `**to_partial_path**`. Update `**app/models/notification.rb**` like this:

app/models/notification.rb

```
def to_partial_path
  'notifications/notification'
end
```

Without this change, collection rendering for notifications would attempt to look up the partial based on each notification's class and we would get an error telling us that no partial named "inbound_email_notifications/inbound_email_notification" exists.

Next, fill in the `notification` partial:

app/views/notifications/_notification.html.erb

```
<li class="p-4">
  <div class="flex space-x-4 items-center">
    <div>
      <button>
        <%= render_inline_svg "check-circle", class:
"h-7 w-7 text-gray-500 hover:text-green-500", title:
"Mark as read" %>
        <span class="sr-only">Mark as read</span>
      </button>
    </div>
    <div>
      <h4 class="text-sm font-medium">
        <%= link_to notification.message,
          notification.url,
          data: {
            id: notification.id,
            action: "click->slideover#open",
            remote: true
          } %>
      </h4>
      <p class="text-sm text-gray-500">
        <%= time_ago_in_words(notification.created_at)
%> ago
      </p>
    </div>
  </div>
</li>
```

Here we are using the `message` and `url` methods defined by each notification subclass to display useful information about the notification to the

user. We also have a button to mark the notification as read that does not do anything yet and we are referencing a `check-circle` svg icon that has not been created.

Create that icon next, from your terminal:

```
touch app/assets/images/check-circle.svg
```

And then fill it in:

app/assets/images/check-circle.svg

```
<svg xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor">
  <path stroke-linecap="round" stroke-linejoin="round"
stroke-width="2" d="M9 12l2 2 4-4m6 2a9 9 0 11-18 0 9 9
0 0118 0z" />
</svg>
```

Finally, update the `NotificationsController` to define the `index` action:

app/controllers/notifications_controller.rb

```
class NotificationsController < ApplicationController
  def index
    @notifications =
current_user.notifications.order(created_at: :desc)
  end
end
```

And update `routes.rb` to add that route to the application:

config/routes.rb

```
resources :notifications, only: %i[index]
```

At this point we have a notification index view ready to display notifications for a user but there is no way for users to see the notifications in the UI. Our next step is adding a notification icon to the authenticated navigation bar and wiring up that icon to open the notifications index page on click.

The first visible piece of notifications in the UI will be an svg icon in the navigation bar, so we'll start there. From your terminal again:

```
touch app/assets/images/bell.svg
```

And then fill that in:

app/assets/images/bell.svg

```
<svg xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor">
  <path stroke-linecap="round" stroke-linejoin="round"
stroke-width="2" d="M15 17h5l-1.405-1.405A2.032 2.032 0
0118 14.158V11a6.002 6.002 0 00-4-5.659V5a2 2 0 10-4
0v.341C7.67 6.165 6 8.388 6 11v3.159c0 .538-.214
1.055-.595 1.436L4 17h5m6 0v1a3 3 0 11-6 0v-1m6 0H9" />
</svg>
```

Heroicons continues to come through with wonderful free icons for us to use.

Next add a partial for the notification indicator. From your terminal:

```
touch app/views/nav/_notifications.html.erb
```

Fill that new partial in like this:

app/views/nav/_notifications.html.erb

```
<div
  id="notifications-container"
  data-controller="dropdown"
  data-action="click@window->dropdown#hide"
```

```
  class="relative"
>
  <button
    id="notifications-menu"
    data-action="click->dropdown#toggle"
    aria-expanded="false"
    class="py-4 px-1 relative border-2 border-
transparent text-gray-800 rounded-full"
  >
    <span class="sr-only">Notifications</span>
    <%= inline_svg_tag('bell', class: "h-6 w-6") %>
    <% if current_user.notifications.unread.exists? %>
      <div class="absolute inset-0 object-right-top -
mr-6">
        <div class="inline-flex items-center px-1.5 py-
0.5 border-2 border-white rounded-full text-xs font-
semibold leading-4 bg-red-500 text-white">
          <%= current_user.notifications.unread.count
%>
        </div>
      </div>
    <% end %>
  </button>
  <div data-dropdown-target="content" class="dropdown-
content absolute w-96 h-96 overflow-y-scroll bg-gray-
100 shadow-lg rounded-md ring-1 ring-black ring-
opacity-5">
    <%= turbo_frame_tag "notifications", src:
notifications_path %>
  </div>
</div>
```

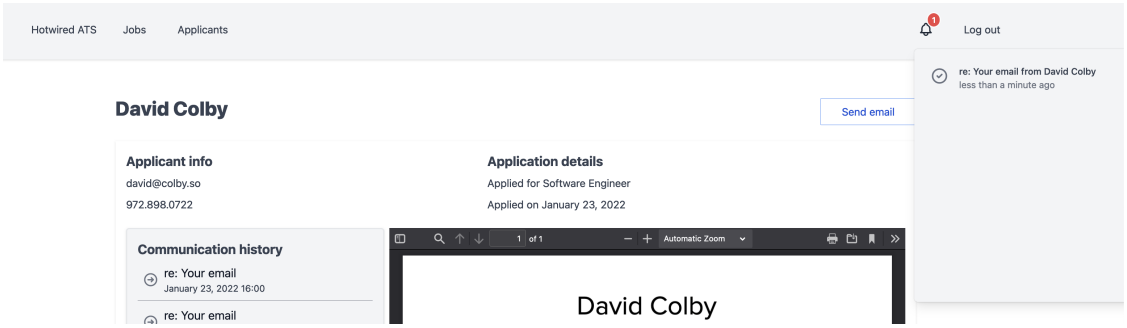
Notice here that we are connecting a dropdown controller with **`data-controller="dropdown"`** and adding actions and targets for that controller. This **`dropdown`** controller does not exist yet. Once we add the controller, the div with the **`dropdown-content`** class will show and hide as the button wrapping the bell icon and the number of notifications is clicked.

Before adding the controller, update the authenticated nav to render this new partial. In **`app/views/nav/_authenticated.html.erb`**:

app/views/nav/_authenticated.html.erb

```
<div class="flex items-center w-full space-x-8">
  <%= link_to "Jobs", jobs_path, class: "rounded px-2 py-1 text-gray-700 hover:text-gray-900 hover:bg-gray-200" %>
  <%= link_to "Applicants", applicants_path, class: "rounded px-2 py-1 text-gray-700 hover:text-gray-900 hover:bg-gray-200" %>
</div>
<div class="flex items-center justify-end px-6 py-4 flex-shrink-0 space-x-8">
  <%= render "nav/notifications" %>
  <%= button_to "Log out", destroy_user_session_path, method: :delete, class: "rounded px-2 py-1 text-gray-700 hover:text-gray-900 hover:bg-gray-200" %>
</div>
```

Now if you refresh the page as a logged in user you will see the bell indicator on the top right of the screen along with the number of unread notifications the user has. Clicking on it will not do anything yet. Next we will build a new Stimulus controller to open and close the notifications menu.



Adding a dropdown menu with Stimulus

First, generate a new Stimulus controller from your terminal:

```
rails g stimulus dropdown
```

And then fill that controller in like this:

app/javascript/controllers/dropdown_controller.js

```
import { Controller } from "stimulus"

export default class extends Controller {
  static targets = ["content"]

  connect() {
    this.open = false
  }

  toggle() {
    if (this.open) {
      this._hide()
    } else {
      this.show()
    }
  }

  show() {
    this.open = true
    this.contentTarget.classList.add("open")
    this.element.setAttribute("aria-expanded", "true")
  }

  _hide() {
    this.open = false
    this.contentTarget.classList.remove("open")
    this.element.setAttribute("aria-expanded", "false")
  }

  hide() {
    if (this.element.contains(event.target) === false
&& this.open) {
      this._hide()
```

```
    }
  }
}
```

This controller expects a `content` target and toggles an `open` class with the `show` and `_hide` functions.

The tricky part is the `hide` function. We do not want the dropdown menu to stay open permanently — any click outside of the menu's content should close the menu. To accomplish this, in the `notifications` partial we added a `data-action` to listen to `click` events on the `window`, calling `dropdown#hide` on each click.

`hide` checks if the click event occurred within the dropdown menu (`this.element`) and if the dropdown menu is open. If the click was outside of the dropdown menu and the menu is open, `_hide` runs and the dropdown is closed.

Next up, the `open` class that we are toggling in the Stimulus controller does not exist yet. Define that class next by creating a new css file from your terminal:

```
touch app/assets/stylesheets/dropdown.css
```

And then update that file:

app/assets/stylesheets/dropdown.css

```
.dropdown-content {
  top: calc(100% - 0.25rem);
  left: 50%;
  transform: rotateX(-90deg) translateX(-50%);
  transform-origin: top;
  opacity: 0.1;
  transition: 280ms all ease-out;
  z-index: 20;
```

```
}
.dropdown-content.open {
  opacity: 1;
  transform: rotateX(0) translateX(-50%);
  visibility: visible;
}
```

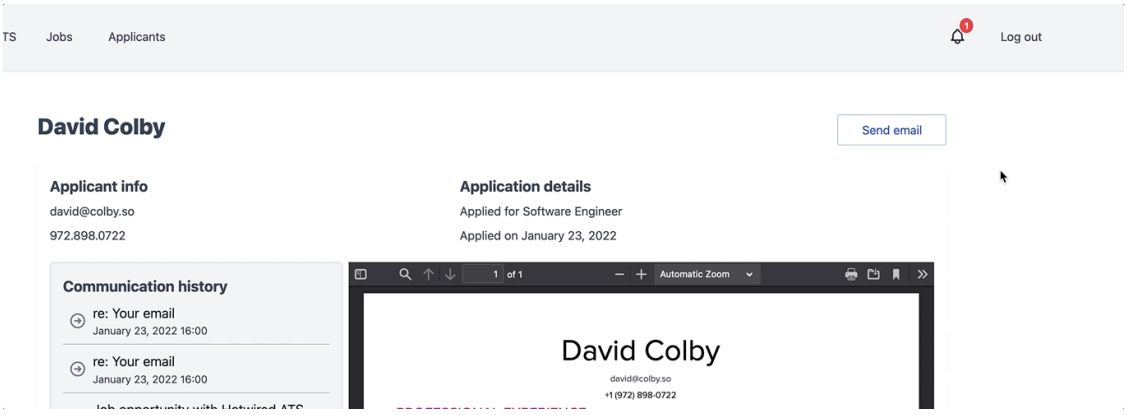
This CSS is responsible for positioning the dropdown content on screen and adding a simple entry and exit animation.

And then import that css into **`app/assets/stylesheets/application.tailwind.css`**:

```
@import "dropdown.css";
```

Now that our Stimulus controller is added along with the supporting CSS to open and close the menu, refresh the page and see that you can click on the bell icon to open and close the notifications dropdown menu.

Now would be a good time to create a few inbound email notifications for the user you are testing with. You can use the **Rails conductor form** for this or (assuming you have already created a few inbound emails), head into the Rails console and run **`Email.inbound.last.create_notification`**



Nice work so far! We have two tasks left in this chapter.

First, we will use Turbo Streams to automatically update the user's notification count each time a new notification is created. Then, we will close the chapter out by allowing users to mark notifications as read with a little bit of StimulusReflex.

Automatic notification broadcasts

In the last chapter we used Turbo Stream broadcasts to prepend new emails to the applicant show page. We will use the same approach to update the unread notification indicator in the navigation bar. Our goal is to keep the count up to date as new notifications come in, without requiring users to refresh the page to see their new notifications.

To do this, start in the `Notification` model:

app/models/notification.rb

```
10  after_create_commit :update_users
11
12  def update_users
13    broadcast_replace_later_to(
14      user,
15      :notifications,
16      target: 'notifications-container',
17      partial: 'nav/notifications',
18      locals: {
19        user: user
20      }
21    )
22  end
```

Here we are using `broadcast_replace_later_to` to replace the content of the notification menu each time a notification is created. The broadcast is sent to the user who owns the notification, ensuring that the correct user receives the update on the front end.

Next, recall that Turbo Stream broadcasts do not have access to session variables like `current_user`. The `notifications` partial currently relies on `current_user` to display the unread notification count — this will not work when we broadcast the partial from the model so we need to update the partial to use a local reference instead.

Head to `app/views/nav/_notifications.html.erb` and update it:

app/views/nav/_notifications.html.erb

```
<div data-controller="dropdown" id="notifications-
container" class="relative" data-action="click@window-
>dropdown#hide">
  <button class="py-4 px-1 relative border-2 border-
transparent text-gray-800 rounded-full"
id="notifications-menu" data-action="click-
>dropdown#toggle" aria-expanded="false">
    <span class="sr-only">Notifications</span>
    <%= inline_svg_tag('bell.svg', class: "h-6 w-6") %>
    <% if user.notifications.unread.exists? %>
      <div class="absolute inset-0 object-right-top -
mr-6">
        <div id="notification-count" class="inline-flex
items-center px-1.5 py-0.5 border-2 border-white
rounded-full text-xs font-semibold leading-4 bg-red-500
text-white">
          <%= user.notifications.unread.count %>
        </div>
      </div>
    <% end %>
  </button>
  <div class="dropdown-content absolute w-96 h-96
overflow-y-scroll bg-gray-100 shadow-lg rounded-md
ring-1 ring-black ring-opacity-5" data-dropdown-
target="content">
    <%= turbo_frame_tag "notifications", src:
notifications_path, autoscroll: true, data: {
autoscroll_block: "start" } %>
```

```
      </div>
    </div>
```

Note that both `user.notifications.read.exists?` and `user.notifications.read.count` now use a local variable, `user`, instead of `current_user`.

Now update the authenticated nav to pass that local variable in when the partial is rendered during page navigation and to subscribe the user to the Turbo Stream notifications channel:

app/views/nav/_authenticated.html.erb

```
6 <%= turbo_stream_from current_user, :notifications %>
7 <%= render "nav/notifications", user: current_user %>
```

With that last change, refresh the page and then create a new inbound email notification for your logged in user and see that the notification indicator updates automatically.

Now let's wrap up our notification system by adding the ability for users to mark notifications as read.

Read notifications

In the `notification` partial, we currently have a Mark as read button rendered alongside each notification, but clicking the button does not do anything yet.

Our aim in this section is to connect that button up to a StimulusReflex action that does two things:

1. Mark the notification as read in the database
2. Remove the notification from the notification list

We will use StimulusReflex to accomplish both of these tasks. To start, generate a new reflex with the `stimulus_reflex` generator. From your terminal:

```
rails g stimulus_reflex Notifications
rails stimulus:manifest:update
```

This generator creates two files — on the server-side `app/reflexes/notifications_reflex.rb` and on the client-side `app/javascript/controllers/notifications_controller.js`. Recall that we need to manually trigger the `stimulus:manifest:update` command after using the StimulusReflex generator.

Begin filling in the notifications controller:

app/javascript/controllers/notifications_controller.js

```
import ApplicationController from
'./application_controller'

export default class extends ApplicationController {
  connect () {
    super.connect()
  }

  read() {
    this.stimulate("Notifications#read", this.element)
  }

  beforeRead(element) {
    element.classList.add("opacity-0")
    setTimeout(() => {
      element.remove()
    }, 150);
  }
}
```

This small Stimulus controller's primary job is to call `this.stimulate` when the `read` action is triggered by a user. Recall from the drag and drop

StimulusReflex implementation that `stimulate` calls a corresponding method on the server which we will add shortly.

The `beforeRead` function takes advantage of the custom **lifecycle callbacks** that StimulusReflex provides to remove the element from the DOM before triggering the reflex. Using this `before` callback allows us to optimistically update the DOM to reflect the user's desired action without waiting on the server to process the change.

Next, update the `NotificationsReflex` like this:

app/reflexes/notifications_reflex.rb

```
class NotificationsReflex < ApplicationReflex
  def read
    notification = element.unsigned[:public]
    notification.read!
    update_notification_count
    morph :nothing
  end

  private

  def update_notification_count
    count = current_user.notifications.unread.count
    if count.positive?
      cable_ready.text_content(selector:
'#notification-count', text: count)
    else
      cable_ready.remove(selector: '#notification-
count')
    end
  end
end
```

The `read` method is the method we call (using `stimulate`) from the Stimulus controller.

In `read`, we first set the notification we are acting on with `element.unsigned[:public]`. StimulusReflex leverages Rails global ids to provide this handy way to access model instances.

Then `notification.read!` sets the notification `read_at` timestamp before `update_notification_count` checks the number of unread notifications the current user has and updates the DOM with the new count. If the user marks their last unread notification as read, `update_notification_count` removes the indicator badge from the DOM entirely instead.

Finally, `morph :nothing` tells StimulusReflex not to run a morph since we handled DOM updates in the `beforeRead` callback and via CableReady with `update_notification_count`.

For this reflex to work, we need to make updates elsewhere in the code base. First, the `read!` method isn't defined in the `Notification` model. Add that now in `app/models/notification.rb`:

app/models/notification.rb

```
28  def read!
29    update_column(:read_at, Time.current)
30  end
```

And because we are referencing `current_user` in the reflex to update the notification count, we need to ensure that StimulusReflex has access to the `current_user` object.

To do this, we will follow the instructions in the wonderful StimulusReflex documentation, starting in `app/channels/application_cable/connection.rb`:

app/channels/action_cable/connection.rb

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
```

```
      self.current_user = find_verified_user
    end

    protected

    def find_verified_user
      if (current_user = env["warden"].user)
        current_user
      else
        reject_unauthorized_connection
      end
    end

  end
end
```

And then delegate calls to `current_user` to `connection` in `app/reflexes/application_reflex.rb`, again lifting straight from the documentation:

app/reflexes/application_reflex.rb

```
class ApplicationReflex < StimulusReflex::Reflex
  delegate :current_user, to: :connection
end
```

With these changes in place, restart your server and then head to `app/views/notifications/_notification.html.erb` to connect the `notifications` Stimulus controller to the DOM:

app/views/notifications/_notification.html.erb

```
    <li
      class="p-4 transition-opacity"
      id="<%= dom_id(notification) %>"
      data-controller="notifications"
      data-public="<%= notification.to_global_id.to_s %>"
    >
      <div class="flex space-x-4 items-center">
```

```
    <div>
      <button data-action="click->notifications#read">
        <%= render_inline_svg "check-circle", class:
"h-7 w-7 text-gray-500 hover:text-green-500", title:
"Mark as read" %>
        <span class="sr-only">Mark as read</span>
      </button>
    </div>
    <div>
      <h4 class="text-sm font-medium">
        <%= link_to notification.message,
          notification.url,
          data: {
            id: notification.id,
            action: "click->slideover#open",
            remote: true
          } %>
      </h4>
      <p class="text-sm text-gray-500">
        <%= time_ago_in_words(notification.created_at)
%> ago
      </p>
    </div>
  </div>
</li>
```

Here on the top level `li` we connected the controller and added the `data-public` attribute that we use in the reflex to reference the correct notification in the database.

Then we added a `data-action` to the Mark as read button, sending clicks on the button to `notifications#read` to remove the notification from the list and mark it as read in the database.

Now, make sure your user has a couple of unread notifications (create a few from the Rails console with `Email.inbound.last.create_notification` if you need to) and then open up the notification menu and click the Mark as read button.

If all has gone well, the notification should transition smoothly out of the DOM and be marked as read in the database. At the same time, the unread count indicator badge should count down each time a notification is read.

To finish up this section (and the chapter!) head over to **`app/controllers/notifications_controller.rb`** to exclude read notifications from the notifications index page — once we remove them from the DOM we do not want users to see them again.

app/controllers/notifications_controller.rb

```
class NotificationsController < ApplicationController
  def index
    @notifications =
current_user.notifications.unread.order(created_at:
:desc)
  end
end
```

Great work in this chapter — we build our own live, reusable notification system with just a little bit of StimulusReflex and a Turbo Stream WebSocket connection.

We have a great base to build new notifications in the future and we have gained an understanding of how a gem like Noticed is constructed so we can more easily adopt it in larger applications after this book is complete.

Before moving on, remember to pause, take a break to reflect, and commit your code if you are following along. You are making great progress so far!

In the next chapter we are going to switch gears a bit and move outside of the administrative interface to build functionality that allows jobs seekers to find open jobs with a company and apply to those jobs.

To see the full set of changes in this chapter, review **this pull request** on Github.