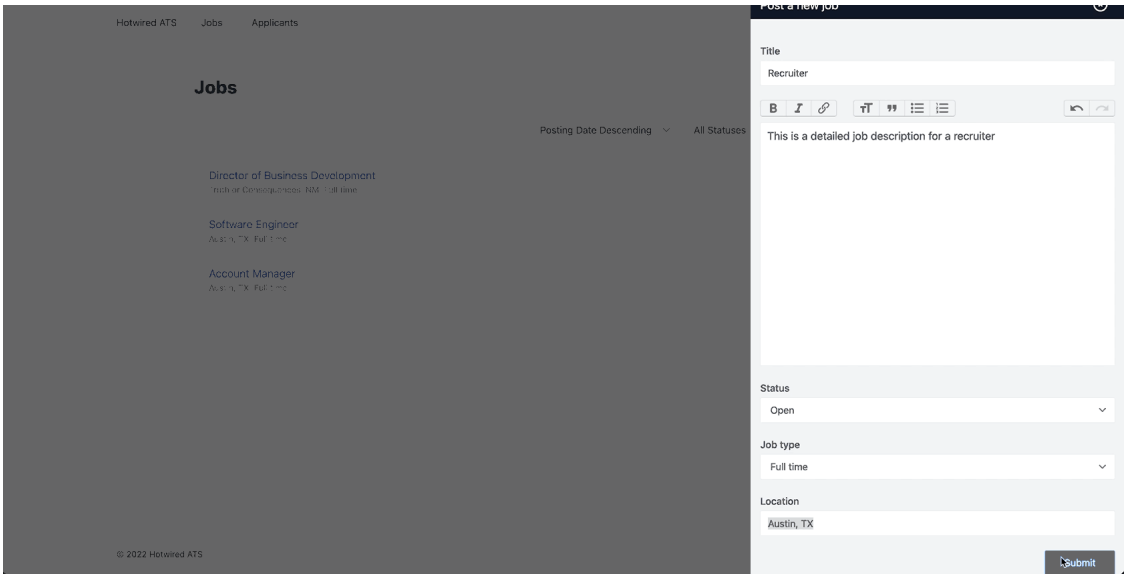
Job Postings

Job postings are a basic building block of an applicant tracking system. When a user needs to hire a new employee, they create a job in their applicant tracking system, advertise that job posting, and receive applications to that job.

Without job postings, an applicant tracking system would just be a database of applications with no way to collect or organize applications in a meaningful way. Since jobs are the key organizational unit of our system, we will build job postings first and then add the ability to apply to those jobs and manage job applicants in future chapters.

When we finish this chapter, logged-in users will have the ability to create new job postings, view those job postings in their account, and edit existing job postings.

To build these features, we will use plenty of standard Rails code with a bit of CableReady (plus a sidebar into Turbo Streams and Turbo Frames), and it will look like this:



Scaffold jobs

To start, use the Rails scaffold generator to create the base model, controller, and views for jobs:

```
rails g scaffold Job title:string status:string:index
job_type:string:index location:string account:references
```

Both job type and job status will be **enums**. Before migrating the database we will update the migration file to prevent null values and set default enum values for new records:

db/migrate/[timestamp]_create_jobs.rb

```
class CreateJobs < ActiveRecord::Migration[7.0]
  def change
    create_table :jobs, id: :uuid do |t|
      t.string :title
      t.string :location
      t.string :status, null: false, default: 'open'
      t.string :job_type, null: false, default:
'full_time'
      t.references :account, null: false, foreign_key:
true, type: :uuid

      t.timestamps
    end
    add_index :jobs, :status
    add_index :jobs, :job_type
  end
end
```

Then migrate the database. From your terminal:

```
rails db:migrate
```

Next, update the job model with relationships, validations, and basic enum definitions:

app/models/job.rb

```
class Job < ApplicationRecord
  belongs_to :account

  validates_presence_of :title, :status, :job_type,
:location

  enum status: {
    draft: 'draft',
    open: 'open',
    closed: 'closed'
  }

  enum job_type: {
    full_time: 'full_time',
    part_time: 'part_time'
  }
end
```

Note that these enums are stored in normal text columns even though, as of Rails 7, Postgres **Enum types** are now **supported natively** in ActiveRecord.

We could store the enums in real enum columns without any extra effort. However, native enums in ActiveRecord have major limitations that make working with them challenging when you expect the enum values to change over time.

Adding new values to an existing enum definition is not yet supported by ActiveRecord, so you still have to drop to raw SQL to add new values. More importantly, you cannot remove a value from an already defined enum without jumping through hoops.

While Postgres Enums are an option, we will use text columns throughout this book.

Next up, define the `has_many` side of the jobs/accounts relationship. In `app/models/account.rb`:

app/models/account.rb

```
class Account < ApplicationRecord
  validates_presence_of :name

  has_many :jobs, dependent: :destroy
  has_many :users, dependent: :destroy
end
```

Job postings are not very useful without a description of the job duties and requirements. You will notice that we did not add a description column to the job in the database. This is because we are going to use **ActionText** to store job descriptions, and users will add job descriptions edit in the UI with **Trix**.

Our use case only needs to support basic styling and links in job descriptions, so Trix, despite its limitations, is a reasonable choice.

Installing ActionText and Trix is straightforward thanks to built-in Rails tasks. From your terminal:

```
rails action_text:install
bundle install
rails db:migrate
```

With ActionText installed, adding a new ActionText-powered field to a model can be done in one line with `has_rich_text`. Update `app/models/job.rb` to add an ActionText-powered field:

app/models/job.rb

```
has_rich_text :description
```

Next up, modify the scaffolded `JobsController` to set the job's account and

app/controllers/jobs_controller.rb

```
  before_action :authenticate_user!

  def create
    @job = Job.new(job_params)
    @job.account = current_user.account

    if @job.save
      redirect_to @job, notice: "Job was successfully
created."
    else
      render :new, status: :unprocessable_entity
    end
  end
```

Here, we first added a `before_action` callback with the Devise provided `authenticate_user!` method.

Then we updated the `create` action to set `job.account` to the current user's account.

At this point, thanks to the Rails scaffold generator, we have a fully functional job resource. We can visit http://localhost:3000/jobs and see all of the jobs in the database, create new jobs, and edit and delete existing jobs. But, we are not using any fun new tools yet — we have standard Rails controller with full page turns for every action. Let's fix that.

In the next section, we will build a nicer-looking jobs index page. Instead of using full page-turns to create job postings, we will use some new magic to create job postings in a drawer that slides out from the side of the screen. I'm excited too.

Viewing and creating job postings

To start, let's update the authenticated nav to link to the jobs index page:

app/views/nav/_authenticated.html.erb

```
<%= link_to "Jobs", jobs_path, class: "rounded px-2 py-1 text-gray-700 hover:text-gray-900 hover:bg-gray-200" %>
```

Then update the job index view:

app/views/jobs/index.html.erb

```
<div class="flex items-baseline justify-between mb-6">
  <h2 class="mt-6 text-3xl font-extrabold text-gray-700">
    Jobs
  </h2>
  <%= link_to "Post a new job", new_job_path, class: "btn-primary-outline" %>
</div>
<div class="shadow overflow-hidden sm:rounded-md">
  <div class="divide-y divide-gray-200">
    <%= render @jobs %>
  </div>
</div>
```

Standard ERB-flavored HTML here that renders a header and a collection of **`@jobs`**, which are set in the **`index`** action in the **`JobsController`**.

Update **`app/views/jobs/_job.html.erb`** next:

app/views/jobs/_job.html.erb

```
<div id="<%= dom_id(job) %>">
  <div class="flex flex-col md:flex-row items-center px-4 py-4 sm:px-6 justify-between">
    <div class="flex-1">
      <%= link_to job.title, edit_job_path(job), class: "text-lg text-blue-600 hover:text-blue-700" %>
      <div class="flex space-x-2 text-gray-500 text-sm">
        <div>
          <%= job.location %>
```

```
          </div>
          <div>
            <%= job.job_type.humanize %>
          </div>
        </div>
      </div>
      <div class="text-right">
        <%= button_to "Delete job", job_path(job),
method: :delete, class: "btn border border-red-200
hover:bg-red-100 text-sm text-red-700", data: {
confirm: "Are you sure?" } %>
      </div>
    </div>
  </div>
```

More standard ERB here, displaying a bit of information about the job, along with links to edit and delete the job posting.

The delete button has a few important items to note. First, we are using the `button_to` helper instead of `link_to`. This is better for accessibility and is semantically the right approach. `link_to` would work fine too, but there is no reason to use a `link_to` for a delete action.

You will notice that we are passing in a `confirm` data attribute to the delete button. Because deleting a record is typically a permanent decision, making users confirm that they want to take this destructive action is a common pattern in web development. When this attribute is present, the text value of the attribute ("Are you sure?") is shown in a modal when the user clicks the link and they need to click the confirm button on the modal to proceed.

Rails provides a simple way to implement this functionality, but how this behavior works has changed in Rails 7 and it is worth taking a brief detour into this topic since the Rails internet is full of questions about these changes.

In Rails 6, `data-confirm` attributes were powered by `@rails/ujs` which shipped by default with Rails. `@rails/ujs` has been deprecated and is no longer included in new Rails 7 applications. Rails now supports `data-confirm`

behavior by default with Turbo; however, the data attribute has changed to `data-turbo-confirm`.

But wait, if `data-turbo-confirm` is the new attribute name, why did we use `data-confirm`? Because we are using **Mrujs**. Mrujs is intended as a direct replacement for `@rails/ujs` and intentionally retained the same attribute names to make migration of existing Rails projects easier. While `data-turbo-confirm` works just fine and we could use it in this project, Mrujs works with `data-confirm` and that is what we will use throughout this project. If you prefer, you can use `data-turbo-confirm` instead.

That is all for the history lesson; back to building jobs.

Creating jobs in a slideover

Currently, clicking on the new job link takes the user to a new page to fill out the job posting form. After they submit the form they are redirected to the job posting. Our goal is to open the job form in a slideover drawer instead of navigating to an entirely new page. When the form is submitted, the newly created job should be added to the existing list of jobs and the slide over drawer should close.

This will be easier than it sounds.

To get started, we need a Stimulus controller to handle opening and closing the slideover. Create that controller from your terminal:

```
rails g stimulus slideover
```

Fill the new Stimulus controller in with:

app/javascript/controllers/slideover_controller.js

```
import { Controller } from 'stimulus'

export default class extends Controller {
```

```
  static targets = [ "slideover" ]

  connect() {
    this.backgroundHtml = this.backgroundHTML()
    this.visible = false
  }

  disconnect() {
    if (this.visible) {
      this.close()
    }
  }

  open() {
    this.visible = true
    document.body.insertAdjacentHTML('beforeend',
this.backgroundHtml)
    this.background =
document.querySelector(`#slideover-background`)
    this.toggleSlideover()
    document.addEventListener("submit:success", () => {
      this.close()
    }, { once: true })
  }

  close() {
    this.visible = false
    this.toggleSlideover()
    if (this.background) {
      this.background.classList.remove("opacity-100")
      this.background.classList.add("opacity-0")
      setTimeout(() => {
        this.background.remove()
      }, 500);
    }
  }

  toggleSlideover() {
    this.slideoverTarget.classList.toggle("right-0")
    this.slideoverTarget.classList.toggle("-right-
```

```
full")
    this.slideoverTarget.classList.toggle("lg:-right-
1/3")
  }

  backgroundHTML() {
    return `<div id="slideover-background" class="fixed
top-0 left-0 w-full h-full z-20"></div>`;
  }
}
```

This Stimulus controller is adapted from **`tailwind-stimulus-components`**, simplified to meet the needs of our application and so we can learn together. In this controller, we are introducing one new Stimulus concept: **targets**.

At the top of the controller, **`static targets = [ "slideover" ]`** defines one target element that our Stimulus controller will rely on. We can reference this target in the controller's methods with **`this.slideoverTarget`**.

Targets are DOM elements that we set when we connect a controller to the DOM. When working with Stimulus, targets are used to obtain a reference to a specific DOM element. In this controller, we use **`this.slideoverTarget`** to toggle classes on the target element. This is a very common use of Stimulus targets.

The event listener that we add when **`open`** is called is the other important piece of the **`SlideoverController`**. This event listener, **`submit:success`**, is how we close the drawer after a successful form submission.

We will see how this works when we update the **`JobsController`** later on in this section. Before that, we need to add the HTML for the slideover, which includes connecting the Stimulus controller to the DOM.

Create a new partial from your terminal:

```
touch app/views/shared/_slideover.html.erb
```

And fill the new partial in with:

app/views/shared/_slideover.html.erb

```
<div data-slideover-target="slideover" class="h-screen
overflow-y-scroll fixed -right-full lg:-right-1/3 top-0
flex-1 flex flex-col w-full lg:w-1/3 bg-gray-100
transition-all duration-500 z-40">
  <div class="flex px-4 justify-between items-center
bg-gray-900">
    <h3 class="text-xl text-white" id="slideover-
header"></h3>
    <button data-action="slideover#close" class="flex
items-center justify-center h-12 w-12 rounded-full
focus:outline-none focus:bg-gray-600 hover:bg-gray-600"
aria-label="Close slideover">
      <%= inline_svg_tag 'close.svg', class: "h-7 w-7
text-white" %>
    </button>
  </div>
  <div class="mt-4 px-4 text-gray-700">
    <div id="slideover-content"></div>
  </div>
</div>
```

Note the `data-slideover-target` on the container element. This data attribute is how we define the `slideoverTarget` in our Stimulus controller. We also have `data-action="slideover-close"` on the close button. Like with the alert in the last chapter, the `close` function in the slideover controller will be called when the button is clicked.

Notice the empty `slideover-content` div. By default, the slideover renders with no content! This is intentional. We are going to reuse this slideover throughout the application, and dynamically insert content each time we use it.

With the partial created, add it to the DOM and connect the Stimulus controller by updating `app/views/layouts/application.html.erb`:

app/views/layouts/application.html.erb

```
<body data-controller="slideover">
  <div class="flex flex-col h-screen justify-between
px-4 md:px-0">
    <%= render "nav/top_nav" %>
    <main class="mb-auto w-full overflow-auto">
      <div class="mx-auto max-w-7xl md:px-6 lg:px-8 py-
8">
        <%= yield %>
      </div>
    </main>
    <%= render "shared/footer" %>
    <div id="flash-container">
      <% flash.each do |key, value| %>
        <%= render "shared/flash", level: key, content:
value %>
      <% end %>
    </div>
  </div>
  <%= render "shared/slideover" %>
</body>
```

Here, we added `` `data-controller="slideover"` `` to the body and inserted the partial just before the end.

Next up, we want to open the content of the new job posting page when a user clicks the Post a new job link on the jobs index page. To do that, we will update the jobs index page like this:

app/views/jobs/index.html.erb

```
<div class="flex items-baseline justify-between mb-6">
  <h2 class="mt-6 text-3xl font-extrabold text-gray-
700">
    Jobs
  </h2>
  <%= link_to "Post a new job",
    new_job_path,
    class: "btn-primary-outline",
    data: {
```

```
      action: "click->slideover#open",
      remote: true
    } %>
</div>
<div class="shadow overflow-hidden sm:rounded-md">
  <div class="divide-y divide-gray-200">
    <%= render @jobs %>
  </div>
</div>
```

Now, our post link has a `data-action` attribute and a `data-remote` attribute. The `data-action` tells Stimulus to fire `slideover#open` when the link is clicked. The `remote` attribute indicates that the link should be handled by Mrujs' CableCar plugin, as described in the **plugin's documentation**.

With our link updated to expect CableCar JSON, we need to update the `JobsController` to respond with the right JSON from the `new` action:

app/controllers/jobs_controller.rb

```
def new
  html = render_to_string(partial: 'form', locals: {
job: Job.new })
  render operations: cable_car
    .inner_html('#slideover-content', html: html)
    .text_content('#slideover-header', text: 'Post a
new job')
end
```

The new action in a standard Rails CRUD controller renders the `new.html.erb` view. We are short-circuiting that default rendering here. Instead, we render the jobs `form` partial to a string and then render `cable_car` operations. These operations are sent back to the browser as a JSON payload that Mrujs handles.

It is important to emphasize here that using CableReady through the CableCar plugin does not require WebSockets or ActionCable, which may be surprising if

you are familiar with CableReady. Using CableCar and Mrujs keeps users in the normal request/response cycle, without involving WebSockets.

The user makes a request in their browser and Rails responds to that request with JSON payload containing CableReady operations. Mrujs receives that response in the browser. Then Mrujs **calls** `cableReady.perform` which applies the operations sent from the server, no WebSockets required.

This CableCar-powered approach will be our primary method of creating, editing, and deleting records throughout this book. We will become very comfortable with CableReady **operations** throughout this book, and we will use CableReady with WebSockets in future chapters.

We will not explore all of the (36!) operations that CableReady supports in this book, but it is important to know that CableReady operations can be sent from almost anywhere and can do almost anything that you want to do on the client — modifying the DOM, logging to the console, playing audio(!), and emitting events.

In this book, we will use CableReady operations for adding, removing, and updating DOM elements and updating data attributes in response to user actions.

For a more detailed introduction, the **CableReady documentation** is a great place to get more familiar with operations and what problems they help solve.

Before we can render operations from a controller, we need to include CableReady::Broadcaster in the controller. Because we will be using this technique in a variety of places in our application, update `app/controllers/application_controller.rb` to include it in all of our controllers:

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  include CableReady::Broadcaster
```

```
    end
```

With CableReady broadcasting operations from the server to the browser and the browser processing those operations, we can pause here to test that the slideover works as expected. Head to **localhost:3000/jobs** and click the Post a job link. You should see the slideover open and populated with the job posting form.

Before moving to handling form submissions, let's pause and make things look a little more pleasant.

We can animate the entry of the slideover with CSS. From your terminal:

```
touch app/assets/stylesheets/animation.css
touch app/assets/stylesheets/slideover.css
```

Add a simple fade animation to **`animation.css`**:

app/assets/stylesheets/animation.css

```
@keyframes fadeIn {
  0% {
    opacity: 0;
  }

  100% {
    opacity: 1;
  }
}
```

And add that animation to the slideover container in **`slideover.css`**:

app/assets/stylesheets/slideover.css

```
#slideover-background {
  background-color: rgba(0, 0, 0, 0.6);
```

```
    animation: fadeIn 0.2s ease-in;
}
```

Import the new stylesheets in `application.tailwind.css`

app/assets/application.tailwind.css

```
@import "animation.css";
@import "slideover.css";
```

Now add some basic style to the job posting form:

app/views/jobs/_form.html.erb

```
<%= form_with(model: job, html: { class: "space-y-6" })
do |form| %>
  <% if job.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(job.errors.count, "error") %>
prohibited this job from being saved:</h2>

      <ul>
        <% job.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="form-group">
    <%= form.label :title %>
    <div class="mt-1">
      <%= form.text_field :title %>
    </div>
  </div>

  <div class="form-group">
    <%= form.rich_text_area :description %>
  </div>
```

```
  <div class="form-group">
    <%= form.label :status %>
    <%= form.select :status,
options_for_select(Job.statuses.map{|key, _value|
[key.humanize, key]}, job.status), {}, { class: "mt-1"
} %>
  </div>

  <div class="form-group">
    <%= form.label :job_type %>
    <%= form.select :job_type,
options_for_select(Job.job_types.map{|key, _value|
[key.humanize, key]}, job.job_type), {}, { class: "mt-
1" } %>
  </div>

  <div class="form-group">
    <%= form.label :location %>
    <div class="mt-1">
      <%= form.text_field :location %>
    </div>
  </div>

  <%= form.submit 'Submit', class: 'btn-primary float-
right' %>
<% end %>
```

Nothing too exciting here. The enum fields are rendered in select tags, and their options are populated using **options_for_select**.

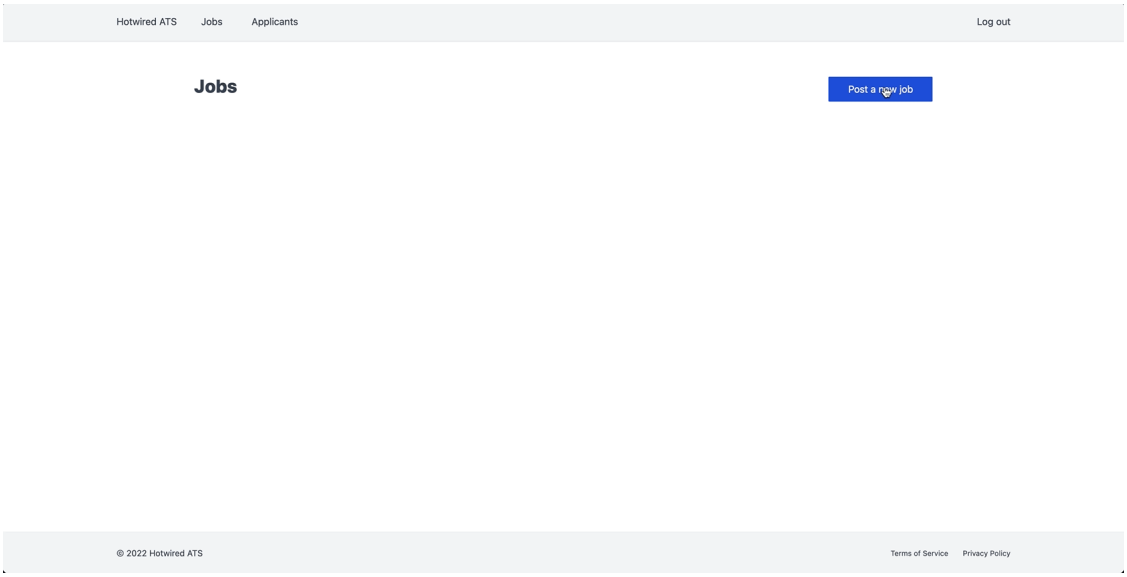
The default Trix editor, rendered with **`form.rich_text_area`**, has options that we do not need to display to our users and looks completely broken. Luckily, we can fix its style and remove the unwanted toolbar options with css. Update **`actiontext.css`** with:

app/assets/stylesheets/actiontext.css

```
@import "trix/dist/trix";
```

```
.trix-button--icon-increase-nesting-level,
.trix-button--icon-decrease-nesting-level,
.trix-button--icon-strike,
.trix-button--icon-code,
.trix-button-group.trix-button-group--file-tools {
display:none; }

trix-editor.trix-content {
  @apply appearance-none w-full max-w-prose bg-white
text-gray-700 border-gray-200 rounded-sm text-lg
focus:ring-1 focus:ring-blue-300 focus:border-blue-300;
  min-height: 400px;
}
```



Now, our form looks much nicer. The slideover neatly transitions in, but when the form is submitted, users are redirected to the jobs show page instead of staying on the job index with the newly created job inserted.

Let's fix that next.

Creating job postings

When a user submits the job posting form, we are going to use CableReady operations to close the slideover drawer and insert the new job into the existing list of jobs. We already have most of the pieces in place for this functionality, but we need to make a few more changes to put it all together, starting in the `JobsController`. Update the `create` action like this:

app/controllers/jobs_controller.rb

```
def create
  @job = Job.new(job_params)
  @job.account = current_user.account
  if @job.save
    html = render_to_string(partial: 'job', locals: {
job: @job })
    render operations: cable_car
      .prepend('#jobs', html: html)
      .dispatch_event(name: 'submit:success')
  else
    html = render_to_string(partial: 'form', locals: {
job: @job })
    render operations: cable_car
      .inner_html('#job-form', html: html), status:
:unprocessable_entity
  end
end
```

We are again rendering a JSON payload that contains CableReady operations for the browser.

When the job is valid and saves successfully, we render the `job` partial to a string and broadcast two CableReady operations to prepend the job to the `jobs` container, and then dispatch a `submit:success` DOM event. Recall that when the slideover opens, our Stimulus controller attaches an event listener for `submit:success` to the document that closes the slideover when it is received:

```
open() {
  document.addEventListener("submit:success", () => {
    this.close()
```

```
  }, { once: true })
}
```

The ability for the server to broadcast arbitrary events that can be picked up by the browser is very powerful, and this feature is one of the reasons why I prefer CableReady over Turbo Streams. We can build this same functionality with Turbo Streams (and I will show you one way to do so in the next section!), but CableReady's flexibility is hard to beat.

It is important to note that we are relying on a custom event, **`submit:success`**, to trigger the slideover close method. Mrujs includes a set of **Ajax lifecycle events** that almost work for our purposes. When the form is submitted successfully, the **`ajax:success`** event is fired automatically. The problem is, the **`ajax:success`** event also fires after the drawer loads the drawer's content in the **`new`** action. If we configured the slideover controller to close on **`ajax:success`**, the slideover would close itself immediately after opening. Not ideal. Firing a custom event when the form submission is successful sidesteps this problem neatly.

When the job is invalid, we re-render the job form partial and update the content of the form to display any validation errors.

Staying in the **`JobsController`**, we also need to update **`job_params`** to allow the **`description`** param through. Without this change, Rails will silently discard whatever the user enters in the description field when saving the job.

app/controllers/jobs_controller.rb

```
def job_params
  params.require(:job).permit(:title, :status,
:job_type, :location, :account_id, :description)
end
```

The changes to the **`JobsController`** action are all we need on the server to properly render the CableReady operations to power the slideover drawer. We

have one last step before we have a fully functional job posting form in the slideover:

Update the job form to add the same `data-remote` attribute that we put on the new job link, and the `#job-form` id that we target when the job submission is invalid:

app/views/jobs/_form.html.erb

```
<%= form_with(
  model: job,
  id: 'job-form',
  html: { class: "space-y-6" },
  data: { remote: true }
) do |form| %>
```

Add the `#jobs` id to the job container div in the jobs index view:

app/views/jobs/index.html.erb

```
<div class="divide-y divide-gray-200" id="jobs">
  <%= render @jobs %>
</div>
```

With those changes in place, you should be able to open and submit the job posting form from the slideover drawer.

When you submit the form with valid information, the drawer will close, and the new job will be prepended to the list of jobs on the index page. Invalid information keeps the drawer open with the errors displayed to the user.

Great work so far!

Next we will add the last new feature of this chapter: editing and deleting job postings with CableReady.

After we add those features, we will conclude the chapter by taking a brief detour to demonstrate how to build this same functionality with Turbo Frames

and Turbo Streams, so you can compare the two approaches.

Slideover edit links

Editing jobs in the slideover is going to feel pretty similar to creating jobs.

First, update the link to the edit page in the job partial with the `data-action` and `remote` attributes:

app/views/jobs/_job.html.erb

```
4  <%= link_to job.title,
5    edit_job_path(job),
6    class: "text-lg text-blue-600 hover:text-blue-700",
7    data: {
8      action: "click->slideover#open",
9      remote: true
10   } %>
```

And then update the `edit` action in the `JobsController`:

app/controllers/jobs_controller.rb

```
23 def edit
24   html = render_to_string(partial: 'form', locals: {
25 job: @job })
26   render operations: cable_car
27     .inner_html('#slideover-content', html: html)
28     .text_content('#slideover-header', text: 'Update
   job')
   end
```

Looks pretty familiar, right?

Now the `update` action:

app/controllers/jobs_controller.rb

```
def update
  if @job.update(job_params)
    html = render_to_string(partial: 'job', locals: {
job: @job })
    render operations: cable_car
      .replace(dom_id(@job), html: html)
      .dispatch_event(name: 'submit:success')
  else
    html = render_to_string(partial: 'form', locals: {
job: @job })
    render operations: cable_car
      .inner_html('#job-form', html: html), status:
:unprocessable_entity
  end
end
```

Again, pretty familiar. Instead of prepending to the list of jobs, we instead use the job's `dom_id` to **replace** the job in the list when the job updates successfully.

Deleting jobs

To remove deleted jobs from the DOM without a page turn, we can again just add the remote attribute and update the controller. In the `job` partial:

app/views/jobs/_job.html.erb

```
<%= button_to "Delete job",
  job_path(job),
  method: :delete,
  class: "btn border border-red-200 hover:bg-red-100
text-sm text-red-700",
  remote: true,
  data: {
    confirm: "Are you sure?"
  } %>
```

And then in the `JobsController`:

app/controllers/jobs_controller.rb

```
61  def destroy
62    @job.destroy
63    render operations: cable_car.remove(selector:
64  dom_id(@job))
    end
```

Use the `remove` operation, target the deleted job's `dom_id`, and we are all set.

Sidebar: Adding jobs with Turbo Streams

We used Stimulus with CableReady and Mrujs to open the slideover, populate the content, and process the form submission. Using Stimulus along with Turbo Streams and Turbo Frames to populate the slideover content and handle the form submission is an alternative.

I find CableReady to be more flexible and more enjoyable to work with in complex situations. However, you can accomplish a very similar result without much more trouble with a Turbo-based approach.

We will start in the Jobs Controller. Update the `new` and `create` actions like this:

app/controllers/jobs_controller.rb

```
  def new
    @job = Job.new
  end

  def create
    @job = Job.new(job_params)
    @job.account = current_user.account
    if @job.save
```

```
      render turbo_stream: turbo_stream.prepend(
        'jobs',
        partial: 'job',
        locals: { job: @job }
      )
    else
      render turbo_stream: turbo_stream.replace(
        'job-form',
        partial: 'form',
        locals: { job: @job }
      ), status: :unprocessable_entity
    end
  end
```

The `new` action has been simplified becase we will render a Turbo Frame in the new action to replace the slideover content instead of broadcasting CableReady operations to update the DOM.

`create` now responds with a Turbo Stream `prepend` action on a successful form submission. This action looks very similar to the equivalent CableReady `prepend` operation and it is powered by the **TurboStreamsTagBuilder**, which is provided by `turbo-rails`.

Turbo Streams can be one of **seven actions**. In addition to `prepend` and `replace` seen here. Available actions are `append`, `update`, `remove`, `before`, and `after`. Multiple Turbo Streams can be rendered in a single response, which allows you to make more sophisticated page updates when needed.

We cannot dispatch DOM events in a Turbo Stream response, so our Stimulus controller will need to be adjusted to close the drawer after the form is submitted. This is an important difference when working with Turbo Streams instead of CableReady. Let's tackle that next.

Update the slideover Stimulus controller like this:

app/javascript/controllers/slideover_controller.js

```
import { Controller } from 'stimulus'

export default class extends Controller {
  static targets = [ "slideover", "form" ]

  connect() {
    this.backgroundHtml = this.backgroundHTML()
    this.visible = false
  }

  disconnect() {
    if (this.visible) {
      this.close()
    }
  }

  open() {
    this.visible = true
    document.body.insertAdjacentHTML('beforeend',
this.backgroundHtml)
    this.background =
document.querySelector(`#slideover-background`)
    this.toggleSlideover()
  }

  close() {
    this.visible = false
    this.toggleSlideover()
    if (this.background) {
      this.background.classList.remove("opacity-100")
      this.background.classList.add("opacity-0")
      setTimeout(() => {
        this.background.remove()
      }, 500);
    }
  }

  toggleSlideover() {
    this.slideoverTarget.classList.toggle("right-0")
    this.slideoverTarget.classList.toggle("-right-
```

```
full")
    this.slideoverTarget.classList.toggle("lg:-right-
1/3")
  }

  backgroundHTML() {
    return `<div id="slideover-background" class="fixed
top-0 left-0 w-full h-full z-20"></div>`;
  }

  handleResponse({ detail: { success } }) {
    if (success) {
      this.formTarget.reset()
      this.close()
    }
  }
}
```

Here, we added a new `form` target, removed the event listener from `open`, and added a new `handleResponse` method at the bottom of the controller:

```
+ static targets = [ "slideover", "form" ]

- document.addEventListener("submit:success", () => {
-  this.close()
- }, { once: true })

+ handleResponse({ detail: { success } }) {
+   if (success) {
+     this.formTarget.reset()
+     this.close()
+   }
+ }
```

We call the `handleResponse` method when the form submission ends. When the form submission is successful, we reset the form to avoid a potential flash

of old content when creating multiple jobs and close the slideover.

When the form submission fails, `handleResponse` does nothing. Because the Turbo Stream response from the server updates the form with validation errors, there is no need for Stimulus to do anything on failed submissions.

Now that the controllers are updated, we will turn to the DOM updates needed to get the slideover working with Turbo.

Update the slideover partial at `app/views/shared/_slideover.html.erb`, replacing the entire content of the file:

app/views/shared/_slideover.html.erb

```
<%= turbo_frame_tag(
  "slideover",
  data: {
    slideover_target: "slideover",
    action: "turbo:submit-end->slideover#handleResponse"
  },
  class: "h-screen overflow-y-scroll fixed -right-full lg:-right-1/3 top-0 flex-1 flex flex-col w-full lg:w-1/3 bg-gray-100 transition-all duration-500 z-50"
) %>
```

The slideover partial will be an empty `<turbo-frame>` element when the page initially loads. Each time we want to use the slideover, we will render a view with a matching `turbo-frame id="slideover"`. Turbo will then automatically replace the content of the slideover frame with the updated content from the server.

Notice here that we set the `data-action` to listen for the `turbo:submit-end` event. Stimulus is not limited to handling `click` events; any emitted event is fair game.

To see this in action, update `app/views/jobs/new.html.erb` like this:

app/views/jobs/new.html.erb

```
<%= turbo_frame_tag "slideover" do %>
  <div class="flex px-6 justify-between items-baseline
bg-gray-900">
    <h3 class="text-xl text-white" id="slideover-
header">Add a new job posting</h3>
    <button data-action="slideover#close" class="flex
items-center justify-center h-12 w-12 rounded-full
focus:outline-none focus:bg-gray-600 hover:bg-gray-600"
aria-label="Close slideover">
      <%= inline_svg_tag 'close.svg', class: "h-7 w-7
text-white" %>
    </button>
  </div>
  <div class="mt-4 px-4 text-gray-700">
    <div id="slideover-content">
      <%= render "form", job: @job %>
    </div>
  </div>
<% end %>
```

We have a matching `slideover` turbo frame plus the full structure of the slideover that previously lived in the `slideover` partial. Since Turbo Frames replace the entire content of the frame, we need our new view to render everything instead of the targeted replacement we did with CableReady.

Update the link to post a new job on the index page:

app/views/jobs/index.html.erb

```
<%= link_to "Post a job",
  new_job_path,
  class: "btn-primary-outline",
  data: {
    action: "click->slideover#open",
    turbo_frame: 'slideover'
  } %>
```

Here, we removed the `remote` attribute, and we added the `turbo_frame` attribute. Because we want the response to the GET request we make to `/jobs/new` to update the content of the `slideover` turbo frame, we need to specify the `turbo_frame` target on the link. This `turbo_frame` attribute is how Turbo connects the dots.

Update the job form to set the `slideover_target`. This change allows Stimulus to reset the form after a successful submission (remember the `formTarget.reset()` call in the handleResponse method that we added to the Stimulus controller).

In `app/views/jobs/_form.html.erb`:

app/views/jobs/_form.html.erb

```
<%= form_with(
  model: job,
  id: 'job-form',
  html: { class: "space-y-6" },
  data: {
    slideover_target: "form"
  }
) do |form| %>
```

With those changes in place, you can refresh the jobs page and see that creating jobs with Turbo Frames and Streams works just as it did with CableReady.

If we want to use Turbo Streams to handle removing deleted jobs from the DOM, we can do that too.

Update the `destroy` action in the `JobsController`:

app/controllers/jobs_controller.rb

```
    def destroy
      @job.destroy
```

```
    render turbo_stream: turbo_stream.remove(@job)
  end
```

Here, we render a Turbo Stream `remove` action, passing in the `job` that we want to remove, instead of the id of an element in the DOM. `turbo-rails` figures out which element we want to remove without needing a specific id.

Update the delete button in the job partial to remove the `remote` attribute.

app/views/jobs/_job.html.erb

```
<%= button_to "Delete job",
  job_path(job),
  method: :delete,
  class: "btn border border-red-200 hover:bg-red-100
text-sm text-red-700",
  data: { confirm: "Are you sure?" }
%>
```

When we click the delete button, the controller will respond with a `<turbo-stream>`, and Turbo will process the DOM updates.

Now that we had a chance to compare approaches, we will move forward in this book using CableReady as the base case. We will create several more slideover drawers in this book, and they will all be built with CableReady. Feel free to use Turbo if you prefer the Turbo-based approach!

We will have many more opportunities to explore both Turbo Frames and Turbo Streams in this book. While CableReady is my preference for this particular interaction, Turbo Frames and Turbo Streams are a better fit in other places in the application and both will play important roles as we add more features to our applicant tracking system.

That is all for this chapter! As usual, this is a good time to pause and review any code that still feels mysterious and take a break to let the new concepts have time to marinate before you move on. If you are coding along with me, this is also a good spot to commit your changes.

Now that our users can create job postings, in the next chapter we will build out the admin interface for creating and managing applicants using some of the techniques we learned in this chapter.

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

Change	Date	PR link
Added a missing `bundle install` command to the ActionText installation instructions.	March 8, 2022	No change to final code.