Public-facing application pages

The last few chapters have been heavy on new content, pushing hard with Turbo, StimulusReflex and CableReady. In this chapter we are going to slow down a bit and practice some of the techniques we have already learned to build public facing job application pages.

At the end of the chapter, we will have used plain-old Rails, CableReady, and Turbo to create an interface that allows job seekers to view open jobs with a company and apply to those jobs.

When we are finished, we will have a basic, public facing careers area with an index of job postings and a job show page that includes an application form to apply to the job.

This chapter is a good opportunity to reinforce some of what we have learned so far while thinking a bit more about the structure of our application and how we can make the application more useful for users in the real world.

As a bonus after we practice our new Turbo and CableReady knowledge, we will close the chapter out by using a brand new StimulusReflex feature to keep the applicants page up to date as new applications are received.

Public-facing careers pages

We will start in the terminal, generating the controllers we will use for the public facing pages.

```
rails g controller Careers
rails g controller Careers::Accounts
```

```
rails g controller Careers::Jobs
```

`Careers::AccountsController` will serve what are commonly called “Careers Pages” in the applicant tracking system world. You have probably visited many applicant tracking system powered-careers pages in your life — when you go to a company’s website and click on “Jobs” or “Careers”, you often end up on a page served by an applicant tracking system.

A careers page is best thought of as a marketing page for the company’s open positions — typically, the careers page includes a list of job postings along with information about working at the company, intended to get people excited about applying.

The `Careers::JobsController` will step in to serve the actual jobs available on the careers page, functioning as both an index route of all job postings for an account and a show route for an individual job posting.

With the controllers created, head to the routes file to add our new routes:

config/routes.rb

```
25  namespace :careers do
26    resources :accounts, only: %i[show] do
27      resources :jobs, only: %i[index show], shallow:
28  true
29    end
    end
```

Here we use a `careers` namespace (matching the controller naming) to group the public facing routes and we avoid deeply nested urls by using the `shallow` option for the jobs routes.

Update the controllers, starting with `app/controllers/careers_controller.rb`:

app/controllers/careers_controller.rb

```ruby
class CareersController < ApplicationController
  layout 'careers'
end
```

As you saw when we updated the routes file, the **`CareersController`** is not serving any routes. Instead, we will use the **`CareersController`** as a base for the controllers in the **`Careers`** namespace, defining a new **`careers`** layout for these controllers.

Before we add the new **`careers`** layout, finish filling in the new controllers. First, **`Careers::AccountsController`**:

app/controllers/careers/accounts_controller.rb

```ruby
class Careers::AccountsController < CareersController
  def show
    @account = Account.find(params[:id])
  end
end
```

Notice that the **`Careers::AccountsController`** inherits from **`CareersController`**, setting the **`AccountsController's`** layout to **`careers`**.

And then **`Careers::JobsController`**:

app/controllers/careers/jobs_controller.rb

```ruby
class Careers::JobsController < CareersController
  before_action :set_account, only: %i[index]
  before_action :set_job, only: %i[show]

  def index
    @jobs = @account.jobs.open.order(title: :asc)
  end

  def show
    @account = @job.account
  end
```

```
13
14    private
15
16    def set_job
17      @job = Job.includes(:account).find(params[:id])
18    end
19
20    def set_account
21      @account = Account.find(params[:account_id])
22    end
  end
```

Here we limit the `@jobs` on the index page to jobs with an `open` status — closed and draft jobs should not receive job applications so we do not want to display them on the careers page.

Note that we are not using the Devise's `authenticate_user!` in the `before_action` in these new controllers. The pages are all intended for public access and should not require a login to access.

Now we will create the new layout for the public-facing careers pages.

From your terminal:

```
touch app/views/layouts/careers.html.erb
app/views/careers/_nav.html.erb
```

And then fill in the new layout:

app/views/layouts/careers.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title).presence || 'Hotwired ATS'
%></title>
    <%= csrf_meta_tags %>
```

```
    <%= csp_meta_tag %>
    <%= action_cable_meta_tag %>

    <%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
    <%= javascript_include_tag "application", "data-turbo-track": "reload", defer: true %>
  </head>

  <body data-controller="slideover">
    <div class="flex flex-col h-screen justify-between px-4 md:px-0">
      <%= render "careers/nav" %>
      <main class="mb-auto w-full overflow-auto">
        <div class="mx-auto max-w-screen-2xl 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>
</html>
```

This layout is very similar to the existing `application` layout, with a different nav bar.

And then the careers `nav` partial:

app/views/careers/_nav.html.erb

```
<nav class="bg-gray-100 shadow">
  <div class="container mx-auto flex text-gray-700 max-w-screen-2xl">
    <div class="flex flex-shrink-0 items-center px-6 py-4">
      <%= link_to "#{@account.name} jobs", careers_account_path(@account), class: "rounded px-2 py-1 text-gray-700 hover:text-gray-900 hover:bg-gray-200" %>
    </div>
  </div>
</nav>
```

Now we need to add views for the **`Career::Accounts`** and **`Career::Jobs`** controllers. From your terminal:

```
touch app/views/careers/accounts/show.html.erb
app/views/careers/jobs/index.html.erb
app/views/careers/jobs/_job.html.erb
app/views/careers/jobs/show.html.erb
```

Starting in **`app/views/careers/accounts/show.html.erb`**:

app/views/careers/accounts/show.html.erb

```
<%= content_for :title, "#{@account.name} Careers" %>
<div class="flex items-baseline justify-between mb-6">
  <h2 class="mt-6 text-3xl font-extrabold text-gray-700">
    Open positions
  </h2>
</div>
<div class="shadow overflow-hidden sm:rounded-md">
  <%= turbo_frame_tag "jobs", class: "divide-y divide-gray-200", src: careers_account_jobs_path(account_id: @account.id), target: "_top" %>
</div>
```

The account show page sets a page title (since “Hotwired ATS” is not a meaningful title for a job seeker) and then displays a list of jobs via the **`turbo_frame_tag "jobs"`**.

In a commercial applicant tracking system, this page would also include information about the company — things like employee testimonials, benefits, and why people love working there. Our application will keep things simple and just list the jobs.

Take note of the **`target: "_top"`** on the **`"jobs"`** Turbo Frame. In previous chapters we set **`target: "_top"`** on each card within the applicants Turbo Frame. Setting **`"_top"`** on a Turbo Frame element means that all navigation within the frame will break out of the frame.

The **`jobs`** frame calls the **`Careers::Jobs#index`** action to populate the frame. Fill the view in:

app/views/careers/jobs/index.html.erb

```
<%= turbo_frame_tag "jobs" do %>
  <%= render @jobs %>
<% end %>
```

Here we are again using collection rendering to render the **`@jobs`**, wrapped in a Turbo Frame that matches the Frame in the account show view.

The **`@jobs`** collection will use the **`job`** partial to render each job as an item in a list of jobs:

app/views/careers/jobs/_job.html.erb

```
    <div>
      <div class="flex flex-col md:flex-row items-center p-
    4 justify-between">
        <div class="md:w-1/2">
          <%= link_to job.title, careers_job_path(job),
    class: "text-lg text-blue-600 hover:text-blue-700" %>
        </div>
```

```
    <div class="md:w-1/3">
      <%= job.job_type.humanize %> in <%= job.location
%>
    </div>
    <div class="text-right">
      <%= link_to "View Details",
careers_job_path(job), class: "btn border border-blue-
200 hover:bg-blue-100 text-sm text-blue-700" %>
    </div>
  </div>
</div>
```

Nothing fancy here.

As mentioned, we do not need to set `target: "_top"` to break out of the frame when a user clicks on a job show page even though these links are wrapped within the `”jobs"` Turbo Frame. Because `target: "_top"` is set on the parent frame, the links will break out by default.

Fill in `app/views/careers/jobs/show.html.erb`:

app/views/careers/jobs/show.html.erb

```
<%= content_for :title, "#{@job.title} at #
{@account.name}" %>
  <div class="max-w-prose mx-auto relative">
    <div class="py-4 text-right">
      <%= link_to "Back to all jobs", :back, class:
"text-lg text-blue-600 hover:text-blue-700" %>
    </div>
    <div class="py-4 px-2 sticky top-0 bg-gray-100">
      <div class="flex justify-between items-baseline">
        <h1 class="text-3xl font-bold pb-2 flex-1"><%=
@job.title %></h1>
        <%= button_to "Apply",
          "#",
          class: "btn-primary-outline disabled:opacity-
75 disabled:hover:cursor-not-allowed",
          id: 'apply-button',
```

```
            remote: true,
            method: :get,
            data: { action: "click->slideover#open" } %>
        </div>
        <p class="text-sm text-gray-500">
          <%= @job.job_type.humanize %> in <%=
@job.location %>
        </p>
      </div>
      <div class="px-2 mt-4">
        <%= @job.description %>
      </div>
    </div>
```

This is standard Ruby and Rails again, plus the now familiar slideover drawer to open up the application form. Instead of a link to open the sildeover, we are using a button because we are going to disable the apply button after a visitor submits an application.

At this point, let's pause and take stock of where we are since we have moved quite quickly through these controllers and views.

So far, we have added two new pages for users to visit — a public careers page. On the careers page, we use a Turbo Frame with an `src` to retrieve a list of job postings for the given account.

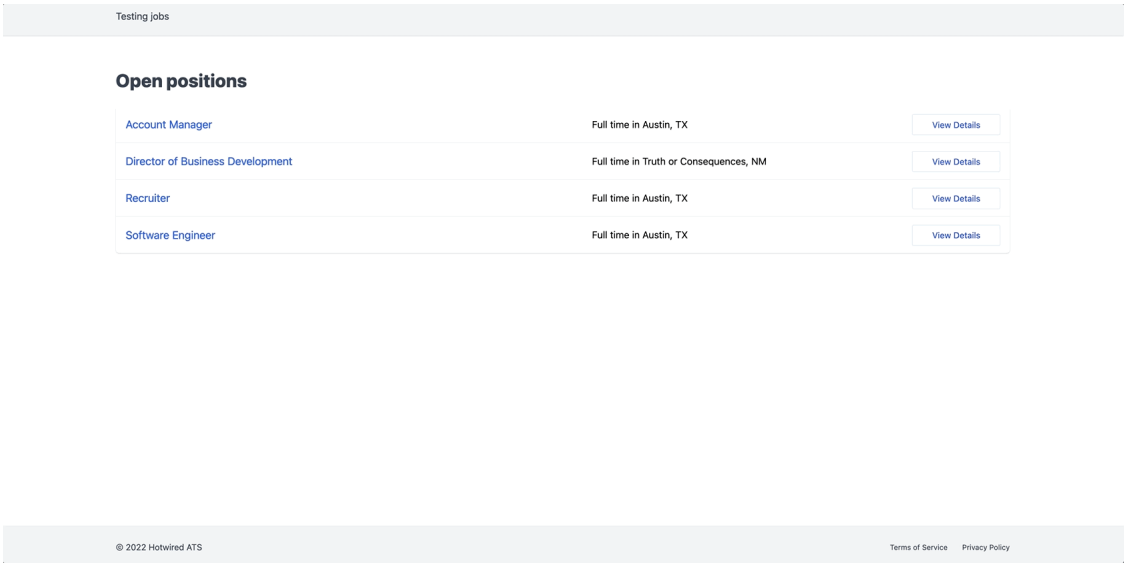
When a user clicks on a job in the list, they navigate to a job show page which displays information about the job and presents a (not yet implemented) application form that will open in a drawer.

To access a careers page yourself, head to **http://localhost:3000/careers/accounts/:id**, swapping in an account id in your test environment with `:id`.

You can also add a temporary link to the Dashboard show page for easy reference:

app/views/dashboard/show.html.erb

```
3 | <%= link_to "Career page",
  careers_account_path(current_user.account) %>
```



Next, let's wire up the Apply button so that visitors can apply to job postings.

Applying to jobs

Visitors will apply to jobs by clicking the Apply button on a job they are interested in and then filling out the form. To accomplish this, we will build a very familiar looking controller.

Get started from your terminal:

```
rails g controller careers::applicants
```

And then fill in `**`Careers::ApplicantsController`**`:

app/controllers/careers/applicants_controller.rb

```
class Careers::ApplicantsController <
ApplicationController
  before_action :set_job
```

```ruby
  def new
    html = render_to_string(partial: 'form', locals: {
applicant: Applicant.new })
    render operations: cable_car
      .inner_html('#slideover-content', html: html)
      .text_content('#slideover-header', text: "#
{@job.title} application")
  end

  def create
    @applicant = Applicant.new(applicant_params)
    @applicant.job = @job
    @applicant.stage = 'application'
    if @applicant.save
      html = render_to_string(partial: 'success',
locals: { applicant: @applicant, job: @job })
      render operations: cable_car
        .inner_html('#slideover-content', html: html)
    else
      html = render_to_string(partial: 'form', locals:
{ applicant: @applicant })
      render operations: cable_car
        .inner_html('#applicant-form', html: html),
status: :unprocessable_entity
    end
  end

  private

  def set_job
    @job = Job.find(params[:job_id])
  end

  def applicant_params
    params.require(:applicant).permit(:first_name,
:last_name, :email, :phone, :resume)
  end
end
```

This code should be feeling routine by now. If it is not, we will have a few more chances to practice it in this book. We are throwing in one new wrinkle — when the application is submitted successfully the drawer stays open and the content of the slideover is replaced with a success message letting the applicant know their application has been submitted.

In a commercial applicant tracking system we might use this success message to share more information about the next steps in the hiring process.

Update `config/routes.rb` to add the `new` and `create` routes:

config/routes.rb

```
namespace :careers do
  resources :accounts, only: %i[show] do
    resources :jobs, only: %i[index show], shallow:
true do
      resources :applicants, only: %i[new create]
    end
  end
end
```

And then create the form and success partial from your terminal:

```
touch app/views/careers/applicants/_form.html.erb
app/views/careers/applicants/_success.html.erb
```

And fill in the form partial:

app/views/careers/applicants/_form.html.erb

```
<%= form_with(model: applicant, url:
careers_job_applicants_path, id: 'applicant-form',
html: { class: "space-y-6" }, data: { remote: true })
do |form| %>
  <% if applicant.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(applicant.errors.count,
```

```
"error") %> prohibited this applicant from being saved:
</h2>

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

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

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

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

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

  <div class="form-group">
    <%= form.label :resume %>
    <%= form.file_field :resume, direct_upload: true,
accept: "application/pdf" %>
  </div>

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

```
      right' %>
      <% end %>
```

Regular old Rails here along with the `data-remote` attribute to submit the form with Mrujs.

And then the success partial with some positive reinforcement for the new applicant.

app/views/careers/applicants/_success.html.erb

```
<div class="space-y-6 flex flex-col">
  <h2 class="text-3xl">Great work, <%=
applicant.first_name %>!</h2>
  <p>Congratulations! Your application for <%=
job.title %> has been submitted. You can close this
drawer when you are done celebrating.</p>
  <button data-action="click->slideover#close"
class="btn-primary-outline">Close this drawer</button>
</div>
```

Now that the controller is created and the routes are defined, we need to update the job show page to add the URL to the application form:

app/views/careers/jobs/show.html.erb

```
<%= button_to "Apply",
  new_careers_job_applicant_path(@job),
  class: "btn-primary-outline disabled:opacity-75
disabled:hover:cursor-not-allowed",
  id: 'apply-button',
  remote: true,
  method: :get,
  data: {
    action: "click->slideover#open"
  } %>
```

Perfect. With all of this in place, we can now visit a careers page for an account with at least one open job, view that job, and then submit an application for that job.

We will add one last small feature to the application process before wrapping up this chapter by broadcasting newly created applications to the authenticated applicants page view.

After an applicant applies to a job, we do not want to encourage them to apply to the same job again. To deter repeated applications, one simple step we can take is disabling the Apply button after an application has been submitted.

In our version of this feature, we will disable the button in the UI after a successful application form submission. This approach is easily defeated by the elite hacker skill of refreshing the page but it is a good start and, more importantly, implementing this feature offers an opportunity to see another CableReady operation in action.

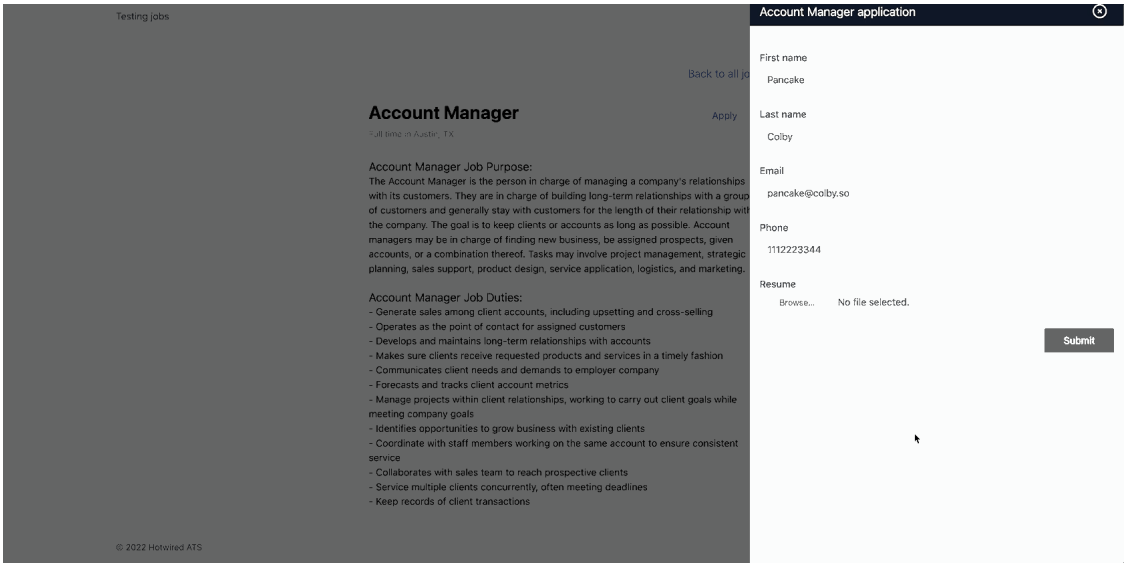
Head back to the **`Careers::Applicants`** controller and update the create action:

app/controllers/careers/applicants_controller.rb

```
15  if @applicant.save
16    html = render_to_string(partial: 'success', locals: {
17  applicant: @applicant, job: @job })
18    render operations: cable_car
19      .inner_html('#slideover-content', html: html)
20      .set_attribute('#apply-button', name: 'disabled',
21  value: '')
22  else
23    html = render_to_string(partial: 'form', locals: {
24  applicant: @applicant })
      render operations: cable_car
        .inner_html('#applicant-form', html: html), status:
    :unprocessable_entity
    end
```

Notice here the new `set_attribute` operation which does what it says on the tin — sets an attribute on the target element.

With this change in place, submit a job application, close the slideover and see that the Apply button is disabled for the user.



Part of what makes CableReady so powerful is the huge set of operations — in a book of this scope we are only scratching the surface of what CableReady can do. It is worthwhile reviewing the **full list of operations** as you spend more time with CableReady.

Broadcasting new applicants

To finish up this chapter, we are going to introduce a brand new piece of functionality in StimulusReflex — reactive broadcasts with `updates_for`.

`updates_for` is conceptually similar to the Turbo Stream model broadcasts that we have used to broadcast new notifications and emails in previous chapters but with a few wrinkles that make `updates_for` more suitable for the task at hand.

Recall that when sending a Turbo Stream broadcast from a model, we do not have access to session variables or any way of knowing what the current state

of the page is. Turbo Stream broadcasts blindly push out a DOM update with local variables and Turbo processes the update for us. Turbo Streams are perfect for simple, predictable updates, such as appending to a single applicant's email list. If a user is on that page, we can assume that update is relevant to them.

Broadcasting new applicants is more complex. Users on the applicants index page may have filters applied that would exclude the new applicant, so inserting a new applicant could make it appear as if the filters are broken. In a more complex application with varying user roles, access levels, and permissions, some users in an account may not have access to the new applicant at all. In more complex scenarios, we could accidentally expose applicants to users that should never have been able to see that applicant.

We cannot safely push an update to everyone in an account at once, we need another way of notifying users that updates are available without just shoving content into the DOM.

This is where **`updates_for`** comes in. Instead of inserting or updating content blindly like we do with a Turbo Stream model broadcast, **`updates_for`** allows us to automatically trigger a seamless reload of the current page. When an update is pushed out, StimulusReflex fetches new content for the page through normal Rails routing (so we have access to session variables and current page context) and morphs in the updated content.

If the new applicant that triggered the update is not relevant to the user's current context, nothing will change. If it is, they will see the new applicant without any disruption to their current page focus through the magic of morphdom.

Let's write some code to see how this works.

First, a small configuration detail. In **`app/models/application_record.rb`**, update it to include **`CableReady::Updatable`**:

app/models/application_record.rb

```ruby
class ApplicationRecord < ActiveRecord::Base
  include CableReady::Updatable
  primary_abstract_class
  self.implicit_order_column = 'created_at'
end
```

We need to add this include before we can use `updates_for` from our models.

Next, head to the applicants index view:

app/views/applicants/index.html.erb

```erb
<div class="flex items-baseline justify-baseline px-10 md:px-6">
  <%= updates_for current_user.account, :applicants do %>
    <%= render "list", grouped_applicants: @grouped_applicants %>
  <% end %>
</div>
```

Here we wrapped the list partial in an `<%= updates_for %>` tag, passing in the channel identifier to listen for updates on. This channel subscription syntax is very similar to what we have done with `<%= turbo_stream_from %>` tags in the application.

An important difference from `<%= turbo_stream_from %>` is that `<%= updates_for %>` must wrap the content you want to update instead of just needing to exist somewhere in the DOM.

Update the `Account` model to enable broadcasts when new applicants are created:

app/models/account.rb

```ruby
has_many :applicants, through: :jobs, enable_updates: { on: :create }
```

`enable_updates` here is how we send updates on model changes to the front end.

It is important to note that we are enabling updates through the `Account` model, which is how the channel identifier is constructed. The first argument to `<%= updates_for %>`, `current_user.account`, is the `belongs_to` side of the relationship, and the second argument, `:applicants`, is the `has_many` side of the relationship.

One last change before we test this out. Because we are broadcasting new applicants when they are created, we do not need to insert them in the `create` action in the `ApplicantsController`.

Update the `create` action to remove the `prepend` operation:

app/controllers/applicants_controller.rb

```
30  def create
31    @applicant = Applicant.new(applicant_params)
32    if @applicant.save
33      render operations: cable_car
34  -     .prepend("#applicants-#{@applicant.stage}", html:
35  html)
36        .dispatch_event(name: 'submit:success')
37    else
38      # snip
39    end
    end
```

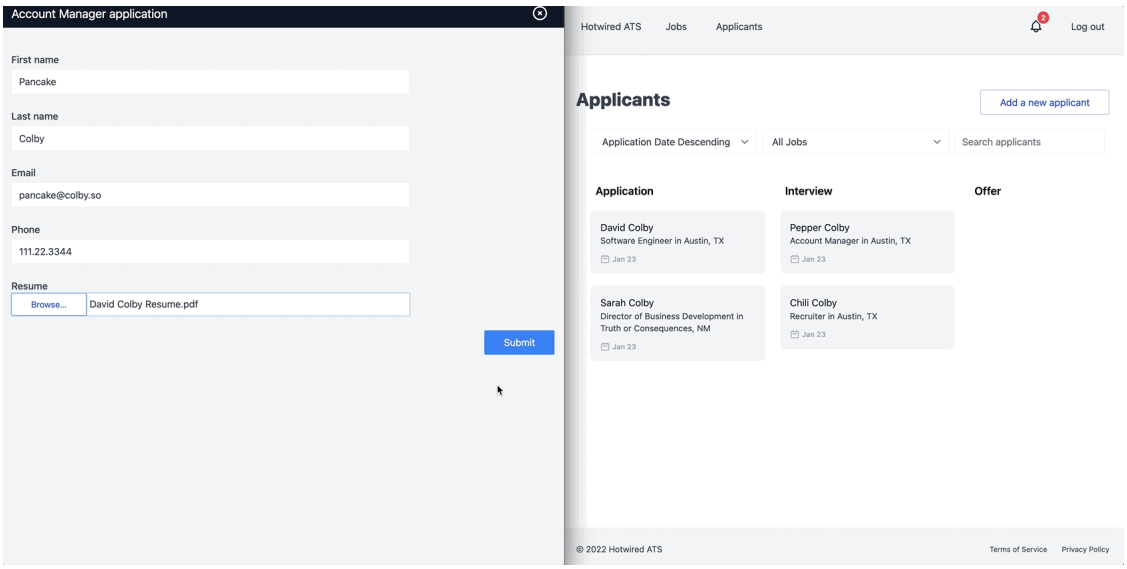
With the controller updated, head to the applicants index page in one browser. In another browser, visit the public facing careers page for the account you are logged in to in the first browser. If all has gone well, you will see the applicant added to the applicants list in the first browser.

Check the Rails server logs as you work and you will see the applicants index controller action running after each applicant creation, seamlessly inserting the newly created applicant without changing anything else on the page. Try it

while focusing a form input or expanding the notifications dropdown and you see that your current page state always stays intact.

```
CableReady::Stream transmitting {"changed"=>["id",
"first_name", "last_name", "email", "phone", "stage",
"job_id", "created_at", "updated_at"]} (via streamed from
gid://test-hotwired-ats/Account/c47ebb79-b4bd-4856-a425-
b1bfd20e9f7b:applicants)
Started GET "/applicants" for 127.0.0.1
```

To see that these updates do not update the page when they should not, try applying a filter that excludes the applicant you are going to create, create the applicant and see the applicant is not added to the page, no extra effort required.



If **`updates_for`** are so much more flexible that Turbo Stream broadcasts, why not just use them everywhere? Why did we use Turbo Stream broadcasts for new emails and notifications?

We are here to learn. Ignoring Turbo Stream model broadcasts entirely would not be useful as a learning exercise. You are likely to encounter Turbo Streams in paid work in the future, so seeing them in action is important.

Part of the motivation for writing this book is to expose people to the wide variety of tools available in the Rails stack, so that when you encounter blockers you feel comfortable looking elsewhere. My philosophy is to cast a wide net, and to not get too attached to over using any one particular tool. There are (almost) always alternatives.

We now have public-facing application pages for users to use to collect applicants to their job postings and we got to learn about the cutting edge `updates_for` functionality provided by StimulusReflex and CableReady. Great work in this chapter!

In the next chapter, we will add another common B2B SaaS app feature — creating and managing users within an account. We will use the next chapter as another chance to sharpen the tools we have learned about so far and to lay the foundation for building an applicant commenting system as we move into the final section of the book.

As usual, this is a great opportunity to commit your code, reflect on what you learned in this chapter, and take a break before moving forward.

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