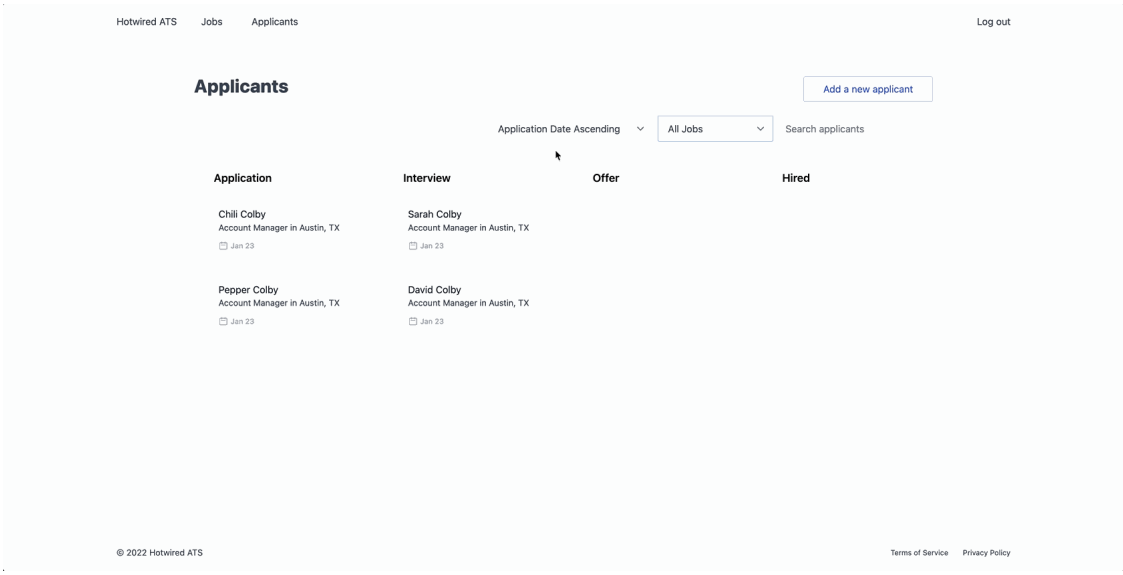
Filtering and sorting with Turbo Frames

One of the most important elements of an applicant tracking system is a robust searching and filtering interface for applicants. Users want to quickly find specific people, see the hiring pipeline for a job, or find their most recent applicants.

Right now, the applicants index page of our application is just a big list of applicants grouped by their hiring stage. In this chapter, we are going to add the ability to search for applicants by name, filter them by the job they applied to, and sort them by their application date.

When we are finished, the applicants index page will work like this:



To accomplish this, we will lean heavily on Turbo Frames to display filtered results. On the backend, we will use **PgSearch** to power applicant search and **Kredis** to store applied filters across multiple filter requests.

When we are finished, we will have a simple, reusable set of filtering tools that we can easily apply to other parts of our application, like the jobs page, with minimal effort.

Add filtering and sorting UI

Before all of the fancy stuff, let's begin with an old-fashioned filter form on the applicants index page. Add this code to the applicants index page, above the list of applicants:

app/views/applicants/index.html.erb

```
<div class="flex mb-6 justify-end">
  <%= form_with url: applicants_path, method: :get,
class: "flex items-baseline" do |form| %>
    <div class="form-group mr-2">
      <%= form.label :sort, class: "sr-only" %>
      <%= form.select :sort,
options_for_select([['Application Date Ascending',
'created_at-asc'], ['Application Date Descending',
'created_at-desc']], params[:sort]) %>
    </div>
    <div class="form-group mr-2">
      <%= form.label :job, class: "sr-only" %>
      <%= form.select :job,
options_for_select(Job.where(account_id:
current_user.account_id).order(:title).pluck(:title,
:id), params[:job]), { include_blank: 'All Jobs' } %>
    </div>
    <div class="form-group mr-2">
      <%= form.label :query, class: "sr-only" %>
      <%= form.text_field :query, placeholder: "Search
applicants", value: params[:query] %>
    </div>
    <%= form.button "Filter", class: "btn-primary" %>
  <% end %>
</div>
```

This is a plain search form, making a GET request to `/applicants` when the form is submitted.

Update the `ApplicantsController` with this ugly but perfectly functional code to make filtering functional:

app/controllers/applicants_controller.rb

```
def index
  if search_params.present?
    @applicants = Applicant.includes(:job)
    @applicants = @applicants.where(job_id:
search_params[:job]) if search_params[:job].present?
    @applicants = @applicants.where('first_name ILIKE ?
OR last_name ILIKE ?', "%#{search_params[:query]}%",
"%#{search_params[:query]}%") if
search_params[:query].present?
    if search_params[:sort].present?
      sort = search_params[:sort].split('-')
      @applicants = @applicants.order("#{sort[0]} #
{sort[1]}")
    end
  else
    @applicants = Applicant.includes(:job).all
  end
end

private

  # Be sure to place this at the bottom of the
  controller, with the other private methods
  def search_params
    params.permit(:query, :job, :sort)
  end
```

This code uses parameters from the search form to filter and order applicant results. When no search parameters are present, the controller returns all applicants.

Besides allowing users to see applicants from all accounts, this filtering mechanism works fine. But this code is hard to read, maintain, and extend. Each filter request also requires a full page turn instead of updating the list of applicants while leaving the rest of the page intact.

Let's take another pass, starting with adding PgSearch to clean up the searching logic.

Add text search with PgSearch

PgSearch is not a new tool and you may have encountered it already in your Rails journey. If PgSearch is new to you, PgSearch is a gem that makes it easier to use Postgres **full text search** in Rails.

PgSearch is extremely powerful and in a commercial application we might consider using it for building a global search interface or for more complex search scenarios. For our learning application, we will use it to search applicants using a **search scope**.

To get started, install PgSearch. From your terminal:

```
bundle add pg_search
```

Restart the Rails application and update the applicant model to include `PgSearch` and add a `text_search` scope:

app/models/applicant.rb

```
include PgSearch::Model
pg_search_scope :text_search,
  against: %i[first_name last_name email],
  using: {
    tsearch: {
      any_word: true,
      prefix: true
```

```
    }
  }
```

Here, we are defining a scope that will search against the first name, last name, and email columns. The `any_word` and `prefix` options make our search a bit more generous by including partial matches and matches against one word when a column has multiple words.

To use this scope in the `ApplicantsController`, update it like this:

app/controllers/applicants_controller.rb

```
10 + @applicants =
11 @applicants.text_search(search_params[:query]) if
   search_params[:query].present?
   - @applicants = @applicants.where('first_name ILIKE ?
   OR last_name ILIKE ?', "%#{search_params[:query]}%",
   "%#{search_params[:query]}%") if
   search_params[:query].present?
```

Now we have a nice neat search scope, but we are still stuffing all of the search logic in the `ApplicantsController` and turning the page on each new filter request.

Let's turn now to building reusable filtering logic with some plain old Ruby and Kredis.

Add filterable concern

In most B2B SaaS applications, you will eventually need to make many resources filterable and searchable, and you do not want to have to rewrite the same logic for every resource.

In this section, we will build a reusable Filterable module that we can include in controllers that need filtering logic. We will first use it in the

`ApplicantsController`. At the end of this chapter, we will reuse the search infrastructure in the `JobsController`.

From your terminal create a new `concern`:

```
touch app/controllers/concerns/filterable.rb
```

Update the new `Filterable` concern like this:

app/controllers/concerns/filterable.rb

```
module Filterable
  def filter!(resource)
    store_filters(resource)
    apply_filters(resource)
  end

  private

  def store_filters(resource)
    session["#{resource.to_s.underscore}_filters"] = {}
unless session.key?("#
{resource.to_s.underscore}_filters")

    session["#
{resource.to_s.underscore}_filters"].merge!
(filter_params_for(resource))
  end

  def apply_filters(resource)
    resource.filter(session["#
{resource.to_s.underscore}_filters"])
  end

  def filter_params_for(resource)
    params.permit(resource::FILTER_PARAMS)
  end
end
```

The `Filterable` `filter!` method takes a `resource` argument. `resource` is a string that maps to a class name, like `"Applicant"`. It then stores the `params` present in the request, and passes those stored filters to the model to query for matching records.

To see how we can use this module, head back to the `ApplicantsController` and update it:

app/controllers/applicants_controller.rb

```rb
class ApplicantsController < ApplicationController
  include Filterable

  def index
    @applicants = filter!(Applicant)
  end
end
```

Here, we included the `Filterable` concern in the controller so that we can use the `filter!` method in this controller, and we updated the `index` action to use the `filter!` method.

Before this will work, the `Applicant` model needs updates too:

1. Define the `filter` method that the `Filterable` module calls in `apply_filters`.
2. Define the scopes used to query the database when filters are applied.
3. Set the `FILTER_PARAMS` that the model will accept.

Head to the `Applicant` model and update it:

app/models/applicant.rb

```rb
    FILTER_PARAMS = %i[query job sort].freeze

    scope :for_job, ->(job_id) { job_id.present? ?
    where(job_id: job_id) : all }
    scope :search, ->(query) { query.present? ?
```

```
37  text_search(query) : all }
38  scope :sorted, ->(selection) { selection.present? ?
39  apply_sort(selection) : all }
40  scope :for_account, ->(account_id) { where(jobs: {
41  account_id: account_id }) }
42
43  def self.apply_sort(selection)
44    sort, direction = selection.split('-')
45    order("applicants.#{sort} #{direction}")
46  end
47
48  def self.filter(filters)
49    includes(:job)
50      .search(filters['query'])
51      .for_job(filters['job'])
52      .sorted(filters['sort'])
    end

    def name
      [first_name, last_name].join(' ')
    end
```

We set the value of `FILTER_PARAMS` to the parameters that we will allow users to filter applicants by, then we added scopes for each filter option (job, text search, and sorting). We also defined two class methods, `filter` and `apply_sort`.

Most of this is standard Rails. `filter` takes the filter parameters passed to it when `filter!` is called in the controller and chains each of the new scopes together.

The scopes check to see if the user cares about filtering by that scope and runs a query if they do. Otherwise, `all` ensures the scope chain continues on. This approach allows users to apply any combination of filters they like without worrying about a nil value in a filter parameter causing unexpected filter results.

With the model updates made, we can go back to the applicants index page, apply some filters, and see that our filters are applied after the page turn.

Before moving on to a more sustainable method for reading and writing applied filters, let's clean up a glaring issue in the code base right now. All users can see all applicants, regardless of which account the applicant belongs to.

To do this, head to the `Applicant` model and add a new scope:

app/models/applicant.rb

```
37 | scope :for_account, ->(account_id) { where(jobs: {
   account_id: account_id }) }
```

Update the `index` action in the `ApplicantsController`:

app/controllers/applicants_controller.rb

```
7 | def index
8 |   @applicants = filter!
9 | (Applicant).for_account(current_user.account_id)
  | end
```

The new `for_account` scope limits applicant results to jobs in the current user's account, preventing data leaks between accounts. We apply this scope outside of the `filter!` code because users cannot apply this filter or otherwise interact with it. Filters in our application can be interacted with while `for_account` exists to wall off data safely.

Go back to the `Filterable` concern where we will look at another method for reading and writing applied filters for a user.

Use Kredis to get and set filters

When a user applies filters, we store the applied filters in the `session`. While this is fine for small scale projects, eventually you will want something more suited to this type of task.

One of the most promising paths for storing this type of ephemeral data is leveraging Redis. A quiet but important change in Rails 7 was including **Kredis** in the default Gemfile (commented out, but present!). Kredis provides a convenient DSL for constructing higher level data structures in a single Redis key, making it simple to use Redis to store interesting but non-permanent data, like the filters that a user currently has applied on a page.

To get started, in your application's Gemfile, uncomment `kredis`:

Gemfile

```
- # gem "kredis"
+ gem "kredis"
```

From your terminal, install Kredis and create a configuration file for Redis:

```
bundle install
mkdir config/redis
touch config/redis/shared.yml
```

And fill in the configuration file with:

config/redis/shared.yml

```
production: &production
  url: <%= ENV.fetch("REDIS_URL", "127.0.0.1") %>
  timeout: 1

development: &development
  host: <%= ENV.fetch("REDIS_URL", "127.0.0.1") %>
  port: <%= ENV.fetch("REDIS_PORT", "6379") %>
  timeout: 1
```

```
test:
  <<: *development
```

Restart the Rails application after completing the installation steps. Now that Kredis is installed, switching from storing filters in the session to storing filters in Kredis is straightforward. Head back to `app/controllers/concerns/filterable.rb` and update it:

app/controllers/concerns/filterable.rb

```
module Filterable
  def filter!(resource)
    store_filters(resource)
    apply_filters(resource)
  end

  private

  def filter_key(resource)
    "#{resource.to_s.underscore}_filters:#
{current_user.id}"
  end

  def store_filters(resource)
    key = filter_key(resource)
    stored_filters = Kredis.hash(key)

stored_filters.update(**filter_params_for(resource))
  end

  def filter_params_for(resource)
    params.permit(resource::FILTER_PARAMS)
  end

  def apply_filters(resource)
    key = filter_key(resource)
    resource.filter(Kredis.hash(key))
```

```
    end
  end
```

The key changes here are using `Kredis.hash` to retrieve filters, and using the `Kredis` `update` method to store new filter options in the Kredis key.

After making these changes visit the applicants index page, submit the filter form, and see that your filters are applied correctly. Neat. Note that storing filters in Kredis makes them semi-permanent. When a user leaves the page and comes back, the application will use their previously applied filters. In some scenarios this is very valuable: Imagine a user clicking on an applicant to view their information and then going back to the applicants index page. In other scenarios, filter persistence may not be necessary and you may wish to keep filters in a more ephemeral store or build a method for easily resetting applied filters.

Clean up applicant group queries

Before we implement partial page updates when filters are applied, let's revisit the ugly, inline erb query we use to group our applicants by stage.

As a reminder, when we built the applicants index page, we added this erb to the view:

app/views/applicants/index.html.erb

```
<% [:application, :interview, :offer, :hired].each do
|key| %>
  <% @applicants.where(stage: key).each do |applicant|
%>
    <%= render "card", applicant: applicant %>
  <% end %>
<% end %>
```

In addition to being clunky, this results in 4 different queries that run each time the index page is loaded. Load the applicants index page in your browser and check the Rails server logs and you will see output like this:

```
SQL (0.7ms)  SELECT "applicants"."id" AS t0_r0,
"applicants"."first_name" AS t0_r1, "applicants"."last_name"
AS t0_r2, "applicants"."email" AS t0_r3,
"applicants"."phone" AS t0_r4, "applicants"."stage" AS
t0_r5, "applicants"."status" AS t0_r6, "applicants"."job_id"
AS t0_r7, "applicants"."created_at" AS t0_r8,
"applicants"."updated_at" AS t0_r9, "jobs"."id" AS t1_r0,
"jobs"."title" AS t1_r1, "jobs"."status" AS t1_r2,
"jobs"."account_id" AS t1_r3, "jobs"."created_at" AS t1_r4,
"jobs"."updated_at" AS t1_r5, "jobs"."location" AS t1_r6,
"jobs"."job_type" AS t1_r7 FROM "applicants" LEFT OUTER JOIN
"jobs" ON "jobs"."id" = "applicants"."job_id" WHERE
"jobs"."account_id" = $1 AND "applicants"."stage" = $2 ORDER
BY applicants.created_at asc  [["account_id", "000"],
["stage", 0]]
Repeat X 4
```

The performance hit will not be noticeable in development, but in a commercial application this request will become a major issue as the database grows. In any applicant tracking system, the most frequently used page is the applicants index page. We do not want that page to be the slowest page because of an inefficient database query!

Let's clean this code up. Update the **`index`** action in the **`ApplicantsController`**:

app/controllers/applicants_controller.rb

```
7  def index
8    @grouped_applicants = filter!(Applicant)
9      .for_account(current_user.account_id)
10     .group_by(&:stage)
11 end
```

`@grouped_applicants` will be a hash, with stage names as the keys and applicants in each stage as the values, like this:

```
{ "offer" => [<Applicant>, <Applicant>], "application" =>
[<Applicant>, <Applicant>], ... }
```

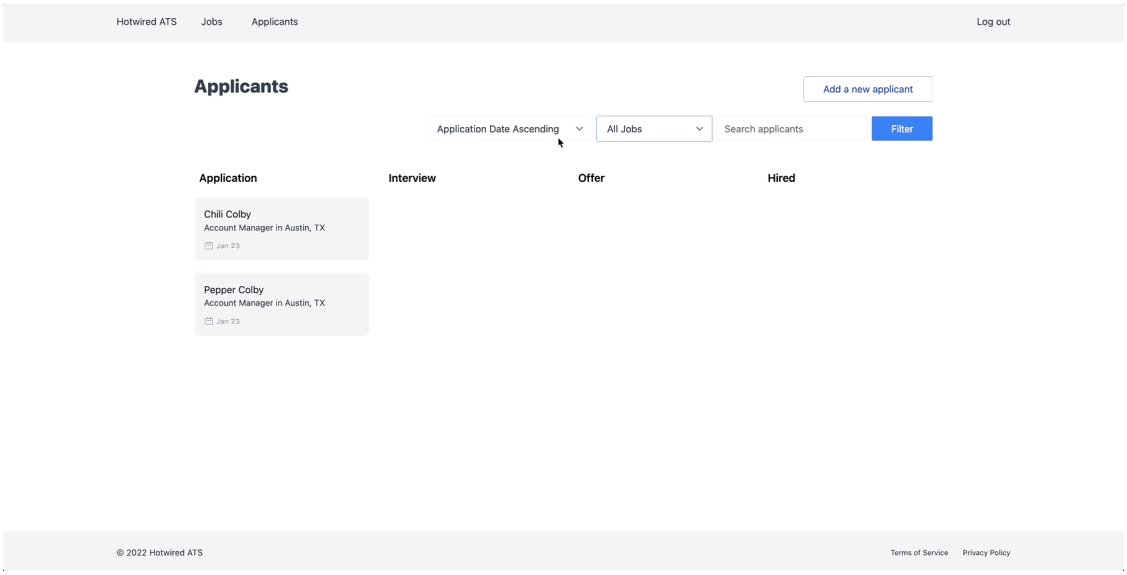
Update the applicants index view to use the new `grouped_applicants` variable:

app/views/applicants/index.html.erb

```
<div class="flex items-baseline justify-between">
  <div class="flex flex-grow mt-4 space-x-6 overflow-
auto" data-controller="drag" data-drag-url-
value="/applicants/:id/change_stage" data-drag-
attribute-value="applicant[stage]">
    <% %w[application interview offer hired].each do
|stage| %>
      <div class="flex flex-col flex-shrink-0 w-72">
        <div class="flex items-center flex-shrink-0 h-
10 px-2">
          <span class="block text-lg font-semibold"><%=
stage.to_s.humanize %></span>
        </div>
        <div id="applicants-<%= stage %>" data-drag-
target="list" data-new-value="<%= stage.to_s %>"
class="h-full">
          <% @grouped_applicants[stage]&.each do
|applicant| %>
            <%= render "card", applicant: applicant %>
          <% end %>
        </div>
      </div>
    <% end %>
  </div>
</div>
```

Note that we are hard coding the order of the applicant stages to ensure that users always see all four stages (even when a stage has zero applicants) in the same order.

Refresh the applicants page and see that everything still works exactly as it did before. To confirm our changes reduced the number of queries on the applicants index page, check the rails server logs after a request to `/applicants`. The repeated `stage` queries should be replaced with a single query to load all applicants displayed on the page.



Filtering the applicants page works well now, but it still requires a full page turn after each filter change. In the next section, we will use Turbo Frames and Stimulus to render partial page updates when filter options change.

Apply filters with Turbo Frames

Turbo Frames are used to scope navigation to a specific part of the page. Using Turbo Frames, we can replace pieces of a page without updating the rest of the page. We took a brief look at Turbo Frames in the sidebar conversation on using Turbo to power the slideover drawer in chapter three. In this section we will take a much deeper dive into how Turbo Frames work.

Turbo Frames are commonly used for building features like in-place editing, tabbed content, lazy loading, and searching, sorting, and filtering data.

In this section, we will update the applicants index page to render the Kanban board inside of a Turbo Frame, and we will update the content of that frame each time the filter submission form changes.

While building this feature, we will also update the filter form to automatically submit when the form changes so that users do not need to manually click a button to submit the form.

To get started, we need a few new view partials. From your terminal:

```
touch app/views/applicants/_list.html.erb
touch app/views/applicants/_filter_form.html.erb
```

We will move code from **`app/views/applicants/index.html.erb`** into these new partials. Start with the **`list`** partial:

app/views/applicants/_list.html.erb

```
<%= turbo_frame_tag "applicants", class: "flex flex-
grow mt-4 space-x-6 overflow-auto", data: { controller:
"drag", drag_url_value: "/applicants/:id/change_stage",
drag_attribute_value: "applicant[stage]" } do %>
  <% %w[application interview offer hired].each do
|stage| %>
    <div class="flex flex-col flex-shrink-0 w-72">
      <div class="flex items-center flex-shrink-0 h-10
px-2">
        <span class="block text-lg font-semibold"><%=
stage.to_s.humanize %></span>
      </div>
      <div id="applicants-<%= stage %>" data-drag-
target="list" data-new-value="<%= stage.to_s %>"
class="h-full">
        <% @grouped_applicants[stage]&.each do
|applicant| %>
```

```
            <%= render "card", applicant: applicant %>
          <% end %>
        </div>
      </div>
    <% end %>
  <% end %>
```

The key change here is replacing the container div with a `<%= turbo_frame_tag %>`. This helper, provided by `turbo-rails` results in a `<turbo-frame id="applicants" …>` when this view is rendered. As we saw in the slideover sidebar, Turbo updates frames based on their ids. When Turbo receives a Turbo Frame response, it expects the body of the response to contain a Turbo Frame with an id that matches the id specified in the request.

Update the `filter_form` partial:

app/views/applicants/_filter_form.html.erb

```
<%= form_with url: applicants_path, method: :get,
class: "flex items-baseline", data: { turbo_frame:
"applicants" } do |form| %>
  <div class="form-group mr-2">
    <%= form.label :sort, class: "sr-only" %>
    <%= form.select :sort,
options_for_select([['Application Date Ascending',
'created_at-asc'], ['Application Date Descending',
'created_at-desc']], params[:sort]) %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :job, class: "sr-only" %>
    <%= form.select :job,
options_for_select(Job.where(account_id:
current_user.account_id).order(:title).pluck(:title,
:id), params[:job]), { include_blank: 'All Jobs' } %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :query, class: "sr-only" %>
    <%= form.text_field :query, placeholder: "Search
```

```
      applicants", value: params[:query] %>
        </div>
        <%= form.button "Search", class: "btn-primary" %>
      <% end %>
```

The only change here is the addition of the `turbo-frame` data attribute, targeting the `applicants` frame that we added to the `list` partial.

This **attribute** tells Turbo that when this form is submitted, the response from the server should update the content of the `applicants` Turbo Frame.

Targeting a Turbo Frame from outside of that frame is a common technique in Turbo, and one that enables Turbo to work well with forms that should not be re-rendered when the Turbo Frame content changes. If the filter form was inside of the `applicants` Turbo Frame, Turbo would re-render the form every time the list of applicants was updated. This re-rendering would cause focus issues and other user experience problems when using the form, especially once the form is automatically submitted with Stimulus later in this chapter.

Next, update the applicants index page to use the new partials.

app/views/applicants/index.html.erb

```
      <div class="flex items-baseline justify-between mb-6">
        <h2 class="mt-6 text-3xl font-extrabold text-gray-
      700">
          Applicants
        </h2>
        <%= link_to "Add a new applicant",
      new_applicant_path, class: "btn-primary-outline", data:
      { action: "click->slideover#open", remote: true } %>
      </div>
      <div class="flex mb-6 justify-end">
        <%= render "filter_form" %>
      </div>
      <div class="flex items-baseline justify-baseline px-10
      md:px-6">
        <%= render "list", grouped_applicants:
```

```
      @grouped_applicants %>
      </div>
```

We now have a Turbo Frame to target, and we have a form targeting that frame when it submits — almost there. Because the form is submitting to the `applicants#index` action, each time the form is submitted, Rails will re-render the content of the entire page on the server.

One way to fix this would be to submit the form to a separate controller action that renders a partial instead of a full page template like the index action does, but that adds extra, unnecessary work that we have to duplicate every time we build a new filter interface like this one.

Instead, we can define a **custom variant** to respond to incoming Turbo Frame requests with appropriate content without a new controller action.

This will be easier than it sounds. It is important to note that the change we are making here is an optimization that is not strictly necessary. When a Turbo Frame request is made, Turbo will process the response HTML, extract the `<turbo-frame id="my_target_frame"` and discard the rest of the content.

I prefer making the extra effort to keep frame updates as efficient as possible. Explicitly responding with Turbo Frame content when a controller action is responsible for full page turns and Turbo Frame requests has an added benefit, to me, of making the code easier to understand when I revisit the code base later.

With that disclaimer out of the way:

First, update the `ApplicationController` to set the request variant to `:turbo_frame` when the request includes a Turbo Frame header, using the `turbo-rails` provided `turbo_frame_request?` **method**.

app/controllers/application_controller.rb

```
      before_action :turbo_frame_request_variant
```

```
6 private
7
8 def turbo_frame_request_variant
9   request.variant = :turbo_frame if
turbo_frame_request?
end
```

On each request, Rails will check to see if a `"Turbo-Frame"` header is present. When the Turbo Frame header is present, the request variant will be set, allowing us to (optionally) render a different view for that request. Next create a `turbo_frame.erb` view from your terminal:

```
touch app/views/applicants/index.html+turbo_frame.erb
```

Update the new Turbo Frame index view:

app/views/applicants/index.html+turbo_frame.erb

```
1 <%= render "list", grouped_applicants:
@grouped_applicants %>
```

Refresh the applicants page, apply some filters and submit the form, and see that the applicants are updated to match the applied filters while the rest of the page remains intact.

One thing you will notice after adding Turbo Frame navigation to the filter form is that the URL no longer updates when the form is submitted. This is by design in Turbo. Frame navigation is scoped to a piece of the page so Turbo defaults to leaving the URL in place when processing a Turbo Frame request, assuming that, from the user's standpoint, they have not navigated to an entirely new page.

This default behavior is usually correct, but not always. Filter forms typically update the URL to make it easy to share a specific set of filters with another

user, or to bookmark those filters in the browser. Updating the URL after a search or filter form submission is a common user experience.

Fortunately for us, a recent change to Turbo made it possible to update the URL of the page from a Turbo Frame request using the `turbo-action` data attribute.

Head back to the `filter_form` partial and update the `form_with`:

app/views/applicants/_filter_form.html.erb

```
<%= form_with url: applicants_path,
  method: :get,
  class: "flex items-baseline",
  data: {
    turbo_frame: "applicants",
    turbo_action: "advance"
  } do |form| %>
```

Now, when the filter form is submitted, the browser's URL will update. Incredible stuff. Let's finish up the filtering experience by making the form automatically submit instead of waiting for the user to click a submit button.

Automatic form submission with Stimulus

We will add this automatic submission functionality with a very small Stimulus controller that we can use to fire a form submission programmatically.

Start by generating a Stimulus controller and add lodash to the project. We will use lodash to debounce events, preventing rapid-fire form submission as form inputs are changed.

In your terminal:

```
rails g stimulus form
yarn add lodash
```

Fill in the new Stimulus controller:

app/javascript/controllers/form_controller.js

```
import { Controller } from "stimulus"
import debounce from "lodash/debounce"

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

  connect() {
    this.submit = debounce(this.submit.bind(this), 200)
  }

  submit() {
    this.formTarget.requestSubmit()
  }
}
```

This controller expects a `form` target in the DOM and uses a `submit` method that calls **requestSubmit** to submit the form after a 200ms debounce.

Update the filter form partial to use this controller to automatically submit the form when inputs are changed:

app/views/applicants/_filter_form.html.erb

```
<%= form_with url: applicants_path,
  method: :get,
  class: "flex items-baseline",
  data: {
    controller: "form",
    form_target: "form",
    turbo_frame: "applicants",
    turbo_action: "advance"
  } do |form| %>
  <div class="form-group mr-2">
    <%= form.label :sort, class: "sr-only" %>
    <%= form.select :sort,
      options_for_select([['Application Date
```

```
Ascending', 'created_at-asc'], ['Application Date
Descending', 'created_at-desc']], params[:sort]),
      {},
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :job, class: "sr-only" %>
    <%= form.select :job,
      options_for_select(Job.where(account_id:
current_user.account_id).order(:title).pluck(:title,
:id), params[:job]),
      { include_blank: 'All Jobs' },
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :query, class: "sr-only" %>
    <%= form.text_field :query, placeholder: "Search
applicants", value: params[:query], data: { action:
"input->form#submit" } %>
  </div>
<% end %>
```

Here, we updated the `<form>` element to add the `data-controller` and `data-form-target` attributes. On each form input we added a `data-action` attribute to call the `submit` function when the input is changed.

We also removed the submit button from the form markup. Since the form is automatically submitted when inputs change, the submit button will not actually do anything. Note that this makes the form unusable if JavaScript is disabled. Users without JavaScript enabled are not who we are building this application for. It is generally safe to assume that the users of this B2B, HR-focused application will have JavaScript enabled on their browser.

For the curious, a less destructive option is to keep the submit button in the UI, and hide it when the `form` controller connects to the DOM, like this:

```
connect() {
  this.submitButtonTarget.remove()
}
```

Add `data-form-target="submit-button"` to the form's submit button. When the Stimulus controller connects, the button disappears, if the controller never connects, the button stays in place.

Cleaning up the filter form

While working through this chapter, you may have noticed a few annoyances with the filter form's code and the experience of using the form.

When constructing the form, we inline the select options of the sort dropdown and the job title filter, which makes the view code a little hard to parse. We can make that a bit better by moving the select options to helper methods.

In the `ApplicantsHelper`:

app/helpers/appplicants_helper.rb

```
module ApplicantsHelper
  def applicant_sort_options_for_select
    [
      ['Application Date Ascending', 'created_at-asc'],
      ['Application Date Descending', 'created_at-
desc']
    ]
  end
end
```

In the `ApplicationHelper`:

app/helpers/application_helper.rb

```
    def job_options_for_select(account_id)
      Job.where(account_id:
```

```
    account_id).order(:title).pluck(:title, :id)
    end
```

Both of these methods are copy pastes of the code previously in the filter form partial. Update the filter form partial now to use those helper methods:

app/views/applicants/_filter_form.html.erb

```
    <%= form_with url: applicants_path, method: :get,
    class: "flex items-baseline", data: { controller:
    "form", form_target: "form", turbo_frame: "applicants",
    turbo_action: "advance" } do |form| %>
      <div class="form-group mr-2">
        <%= form.label :sort, class: "sr-only" %>
        <%= form.select :sort,

    options_for_select(applicant_sort_options_for_select,
    params[:sort]),
          {},
          { data: { action: "change->form#submit" } }
        %>
      </div>
      <div class="form-group mr-2">
        <%= form.label :job, class: "sr-only" %>
        <%= form.select :job,

    options_for_select(job_options_for_select(current_user.ac
    params[:job]),
          { include_blank: 'All Jobs' },
          { data: { action: "change->form#submit" } }
        %>
      </div>
      <div class="form-group mr-2">
        <%= form.label :query, class: "sr-only" %>
        <%= form.text_field :query, placeholder: "Search
    applicants", value: params[:query], data: { action:
    "input->form#submit" } %>
```

```
      </div>
    <% end %>
```

A small, but useful change for readability. The `job_options_for_select` method will be reused in other places later. A list of job titles in a select menu is a common need in the application.

The next clean up task is ensuring that the default value of form inputs always matches the current state of the filter values stored in the Kredis filters hash.

This issue is noticeable if you apply filters to the applicants page and then leave the page and come back. The list of applicants will be filtered based on the last applied filters were, but those filters will not be visible to the user.

We can fix this issue by updating the form inputs to have a value when they are first rendered. To do this, add a helper to the `ApplicationHelper`:

app/helpers/application_helper.rb

```
21 def fetch_filter_key(resource, user_id, key)
22   Kredis.hash("#{resource}_filters:#{user_id}")[key]
23 end
```

This helper is a convenience method for accessing the applied filter for a specific user and resource combination. To use it, update the filter form again:

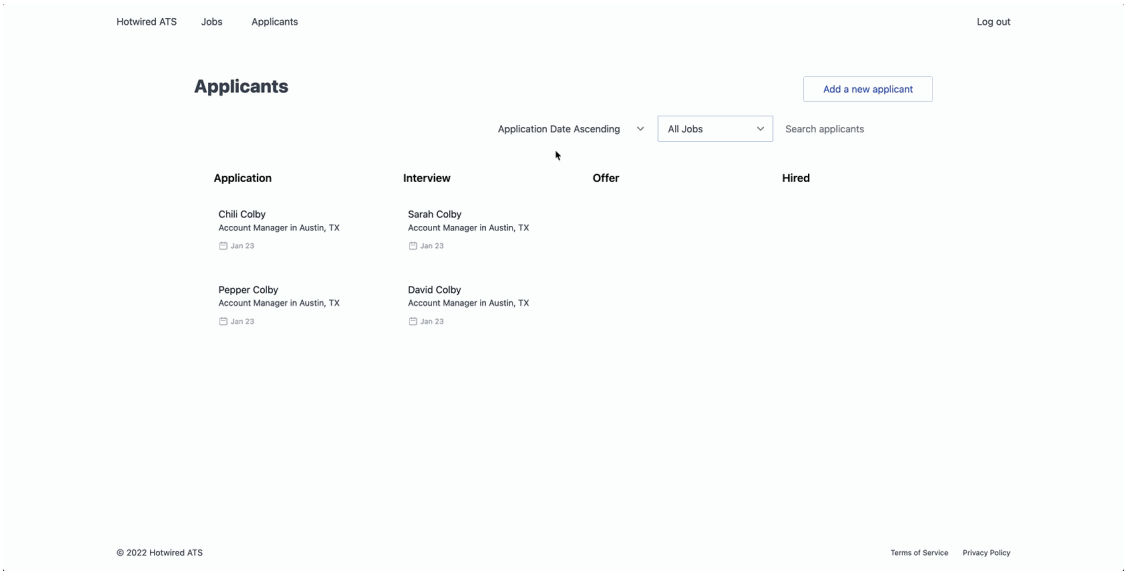
app/views/applicants/_filter_form.html.erb

```
    <%= form_with url: applicants_path, method: :get,
    class: "flex items-baseline", data: { controller:
    "form", form_target: "form", turbo_frame: "applicants",
    turbo_action: "advance" } do |form| %>
      <div class="form-group mr-2">
        <%= form.label :sort, class: "sr-only" %>
        <%= form.select :sort,
          options_for_select(
            applicant_sort_options_for_select,
            fetch_filter_key("applicant", current_user.id,
```

```
"sort")
      ),
      {},
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :job, class: "sr-only" %>
    <%= form.select :job,
      options_for_select(
        job_options_for_select(current_user.account),
        fetch_filter_key("applicant", current_user.id,
"job")
      ),
      { include_blank: 'All Jobs' },
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :query, class: "sr-only" %>
    <%= form.text_field :query, placeholder: "Search
applicants", value: fetch_filter_key("applicant",
current_user.id, "query"), data: { action: "input-
>form#submit" } %>
  </div>
<% end %>
```

Now, if filter values are applied from the Kredis store on the initial page load, they will be visible when the form renders.

Nice work! We now have a fully functional filtering form on the applicants page, updating the applicants board almost instantly in response to user input. If all has gone according to plan, your applicants page should look like this:



Besides being efficient and fairly easy to reason about, this filtering code is flexible enough to be used throughout our application. To demonstrate that, let's wrap up this chapter by adding filtering to the jobs index page.

Filtering jobs

To add filtering to jobs, the primary tasks will be adding a form to the jobs index page and updating the job model to support the `Filterable` concern we built at the start of this chapter.

Start by updating the jobs index page. From your terminal, add the new partials:

```
touch app/views/jobs/index.html+turbo_frame.erb
app/views/jobs/_filter_form.html.erb
app/views/jobs/_list.html.erb
```

Fill in the jobs filter form partial:

app/views/jobs/_filter_form.html.erb

```
<%= form_with url: jobs_path, method: :get, class:
"flex items-baseline", data: { controller: "form",
form_target: "form", turbo_frame: "jobs", turbo_action:
```

```
"advance" } do |form| %>
  <div class="form-group mr-2">
    <%= form.label :sort, class: "sr-only" %>
    <%= form.select :sort,
      options_for_select(job_sort_options_for_select,
fetch_filter_key("job", current_user.id, 'sort')),
      {},
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :status, class: "sr-only" %>
    <%= form.select :status,
      options_for_select(status_options_for_select,
fetch_filter_key("job", current_user.id, 'status')),
      { include_blank: 'All Statuses' },
      { data: { action: "change->form#submit" } }
    %>
  </div>
  <div class="form-group mr-2">
    <%= form.label :query, class: "sr-only" %>
    <%= form.text_field :query, placeholder: "Search
jobs", value: fetch_filter_key("job", current_user.id,
'query'), data: { action: "input->form#submit" } %>
  </div>
<% end %>
```

Like in the applicants filter form, we set a `turbo-frame` data attribute, attach the `form` controller, and set the form inputs to automatically submit the form when they change.

We are using two helper methods for the select dropdowns. Add those to the `JobsHelper` module next:

app/helpers/jobs_helper.rb

```
module JobsHelper
  def job_sort_options_for_select
    [
```

```
      ['Posting Date Ascending', 'created_at-asc'],
      ['Posting Date Descending', 'created_at-desc'],
      ['Title Ascending', 'title-asc'],
      ['Title Descending', 'title-desc']
    ]
  end

  def status_options_for_select
    Job.statuses.map{|key, _value| [key.humanize, key]}
  end
end
```

Now fill in the **`list`** partial:

app/views/jobs/_list.html.erb

```
<%= turbo_frame_tag "jobs", class: "divide-y divide-gray-200" do %>
  <%= render jobs %>
<% end %>
```

And update **`app/views/jobs/index.html.erb`** to use the new partials

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: "mouseup->slideover#open", remote: true } %>
</div>
<div class="flex mb-6 justify-end">
  <%= render "filter_form" %>
</div>
<div class="shadow overflow-hidden sm:rounded-md">
```

```
  <%= render "list", jobs: @jobs %>
</div>
```

Fill in the Turbo Frame index view:

app/views/jobs/index.html+turbo_frame.erb

```
<%= render "list", jobs: @jobs %>
```

This should look pretty familiar so far! Update the `JobsController` to filter jobs next:

app/controllers/jobs_controller.rb

```
include Filterable

def index
  @jobs = filter!(Job).for_account(current_user.account_id)
end
```

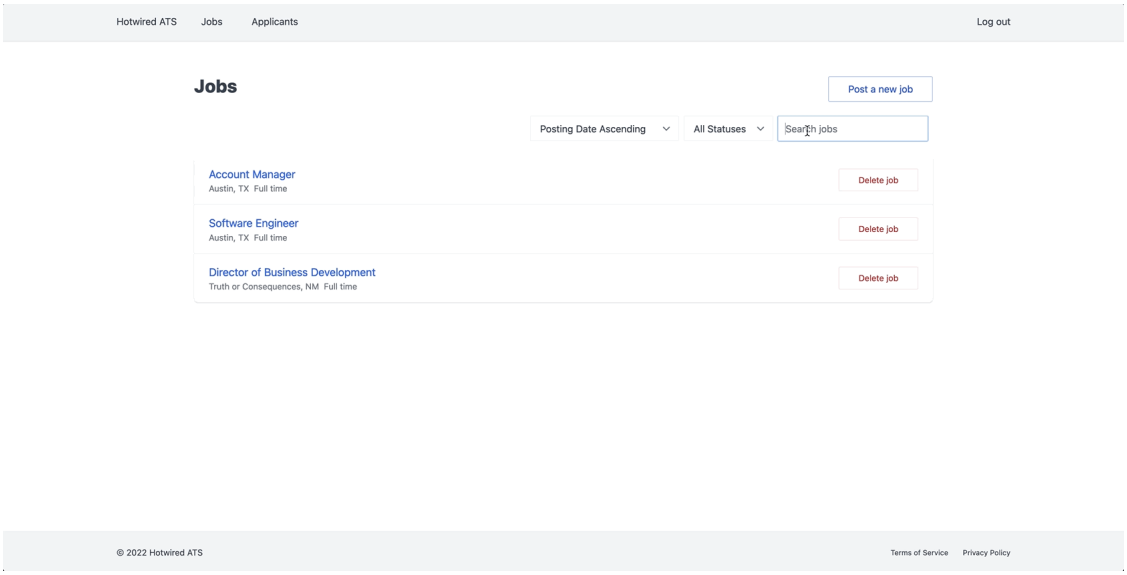
Update the job model to add the scopes and `Filterable` compatibility:

app/models/job.rb

```
FILTER_PARAMS = %i[query status sort].freeze

scope :for_account, ->(account_id) { where(account_id: account_id) }
scope :for_status, ->(status) { status.present? ? where(status: status) : all }
scope :search_by_title, ->(query) { query.present? ? where('title ILIKE ?', "%#{query}%") : all }
scope :sorted, ->(selection) { selection.present? ? apply_sort(selection) : all }

def self.filter(filters)
  search_by_title(filters['query'])
```

```
    .for_status(filters['status'])
    .sorted(filters['sort'])
end

def self.apply_sort(selection)
  return if selection.blank?

  sort, direction = selection.split('-')
  order("#{sort} #{direction}")
end
```

Et voila! Head to the jobs page, apply some filters, and see them update the list of jobs.



Great work in this chapter! We got our first real taste of Turbo Frames, we were introduced to working with ephemeral data in Kredis, and we built reusable code to handle filtering resources throughout our application.

As always, before moving on, this is a great time to commit your code and take a break to reflect on any new concepts you encountered in this chapter.

In the next chapter, we will move from the applicants index page to the applicants show page, building a basic applicant profile layout and adding the

ability to send emails to applicants and receive replies to those emails using `**ActionMailbox**`.

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