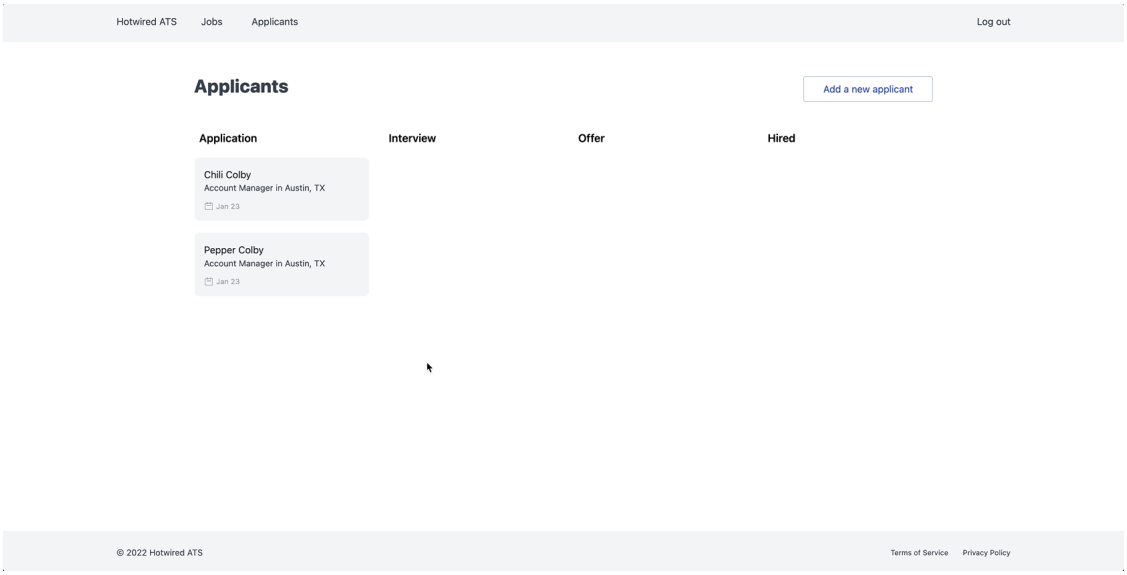
Creating and moving applicants

Companies use applicant tracking systems to attract candidates and manage their hiring process, not to create job postings in an administrative interface.

Right now our version of an ATS is a list of jobs, posted into the void, not very useful. We will make the application more valuable in this chapter by creating an `Applicants` resource and then adding the ability for users to create and manage applicants manually in the admin interface.

When we finish this chapter, we will have a Kanban board styled `Applicants` page. Applicants will be grouped by hiring stage and users will be able to drag-and-drop applicants between stages. We will use the same CableReady-powered slideover drawer to add new applicants, and we will explore two different methods for dragging applicants between stages: a Stimulus-only version and a StimulusReflex version.

At the end of this chapter, the applicants page will look like this:



Build applicant resource

To begin, generate a scaffold from your terminal and migrate the database:

```
rails g scaffold Applicant first_name:string
last_name:string email:string:index phone:string
stage:string:index status:string:index job:references
rails db:migrate
```

Update the applicant model at **`app/models/applicant.rb`** with enum definitions, basic validations, and a **`name`** helper method:

app/models/applicant.rb

```
class Applicant < ApplicationRecord
  belongs_to :job

  enum stage: {
    application: 'application',
    interview: 'interview',
    offer: 'offer',
    hired: 'hire'
  }

  enum status: {
```

```
    active: 'active',
    inactive: 'inactive'
  }

  validates_presence_of :first_name, :last_name, :email

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

We are storing an applicant's hiring stage directly on the applicants table in our application. If we were building an application intended for commercial use, a better approach would be moving hiring stages to a separate database table to allow each company to create their own hiring stages. The four static stages we are using will work just fine for the learning application that we are building.

Update the job model to add the applicants **`has_many`** association:

app/models/job.rb

```
has_many :applicants, dependent: :destroy
```

Add a link to the applicants index page to the authenticated navigation bar:

app/views/nav/_authenticated.html.erb

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

Fill in **`app/views/applicants/index.html.erb`** to create a basic Kanban layout:

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 items-baseline justify-between">
  <div class="flex flex-grow mt-4 space-x-6 overflow-
auto">
    <% [:application, :interview, :offer, :hired].each
do |key| %>
      <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"><%=
key.to_s.humanize %></span>
        </div>
        <div id="applicants-<%= key %>" class="h-full">
          <% @applicants.where(stage: key).each do
|applicant| %>
            <%= render "card", applicant: applicant %>
          <% end %>
        </div>
      </div>
    <% end %>
  </div>
</div>
```

Here, we loop through groups of applicants based on their hiring stage and render each applicant.

For simplicity, we are querying for applicants by stage in the view which is not efficient or sustainable, but it works for now. We will improve this grouping related code in a future chapter of this book.

Take note of the **`applicants-#{key}`** id assigned to each applicant group. We will use this id to insert newly created applicants into the correct location on the board.

Each applicant will be rendered as a card on the board, and we will render them using a `card` partial.

Create that partial from your terminal:

```
touch app/views/applicants/_card.html.erb
```

Fill the new card partial in:

app/views/applicants/_card.html.erb

```
<div class="flex flex-col pb-2 overflow-auto">
  <div class="relative flex flex-col items-start p-4 mt-3 bg-gray-100 rounded-lg cursor-move bg-opacity-90 group hover:bg-opacity-100">
    <h4 class="text-gray-900"><%= link_to applicant.name, applicant, class: "cursor-pointer" %></h4>
    <p class="text-sm text-gray-700"><%= applicant.job.title %> in <%= applicant.job.location %></p>
    <div class="flex items-center w-full mt-3 text-xs font-medium text-gray-400">
      <div class="flex items-center" title="Application Date">
        <%= inline_svg_tag 'calendar.svg', class: 'h-4 w-4 inline-block' %>
        <span class="ml-1 leading-none"><%= l(applicant.created_at.to_date, format: :short) %></span>
      </div>
    </div>
  </div>
</div>
```

The card partial is rendering a calendar svg icon that we have not added yet. Create it from the terminal:

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

Fill the new icon file in:

app/assets/images/calendar.svg

```
<svg xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor">
  <path stroke-linecap="round" stroke-linejoin="round"
stroke-width="2" d="M8 7V3m8 4V3m-9 8h10M5 21h14a2 2 0
002-2V7a2 2 0 00-2-2H5a2 2 0 00-2 2v12a2 2 0 002 2z" />
</svg>
```

With these changes in place, the Applicants index page now renders, but we cannot create new applicants through the UI so the page is not very useful yet. Next up we will use the slideover drawer technique we used in the last chapter to enable users to create applicants in the UI.

Create applicants

Update the `new` action in the `ApplicantsController`:

app/controllers/applicants_controller.rb

```
before_action :authenticate_user!

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: 'Add an
applicant')
end
```

Here, we added `authenticate_user!` to require users to login before accessing these routes and then we updated the `new` action to return

CableReady JSON. This code is nearly identical to the `new` action in the `JobController`.

Now update the applicants form partial:

app/views/applicants/_form.html.erb

```
<%= form_with(model: applicant, 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 :stage %>
    <%= form.select :stage,
options_for_select(Applicant.stages.map{ |key, _value|
[key.humanize, key]}, applicant.stage), {}, { class:
"mt-1" } %>
  </div>

  <div class="form-group">
    <%= form.label :job_id %>
    <%= form.select :job_id,
options_for_select(Job.where(account_id:
current_user.account_id).order(:title).pluck(:title,
:id)), {}, { class: "mt-1" } %>
  </div>

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

When we click the Add a new applicant button the applicant form will open in a slideover, but submitting the form will redirect to the applicant show page.

When a new applicant is created the slideover should close and the applicant should be prepended to the applicants list, mirroring the functionality on the jobs page.

Head back to the `ApplicantsController` and update the `create` action:

app/controllers/applicants_controller.rb

```
def create
  @applicant = Applicant.new(applicant_params)
  if @applicant.save
    html = render_to_string(partial: 'card', locals: {
```

```
32  applicant: @applicant })
33      render operations: cable_car
34        .prepend("#applicants-#{@applicant.stage}", html:
35  html)
36        .dispatch_event(name: 'submit:success')
37    else
38      html = render_to_string(partial: 'form', locals: {
39  applicant: @applicant })
        render operations: cable_car
          .inner_html('#applicant-form', html: html),
      status: :unprocessable_entity
    end
  end
```

As in the `new` action, `create` will look familiar. Like the `JobsController`, on a successful submission the newly created resource is prepended to a target element in the DOM, and the slideover is closed via the `submit:success` DOM event.

In all applicant tracking systems, applicants can upload a resume with their application, so we will add that feature next. To support resume uploading, we will install and configure **ActiveStorage** in our application.

ActiveStorage resumes

Our goal is to allow a resume file to be attached to an applicant when the applicant is created. To start, update the `Applicant` model to tell ActiveStorage about the `resume` attachment:

app/models/applicant.rb

```
18  has_one_attached :resume
```

`has_one_attached` defines a single `resume` attribute that will store the attached file. If applicants were allowed to have multiple resumes, we could use **has_many_attached** instead.

With `resume` added to the applicant model, we can update the applicants form partial to add a new file upload field:

app/views/applicants/_form.html.erb

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

Update `applicant_params` in the `ApplicantsController` to add the new `resume` attribute:

app/controllers/applicants_controller.rb

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

At this point refresh the page, open the applicant creation slideover, and see that the resume field is visible on the form. Attach a pdf and save the applicant and the pdf will be attached to the applicant.

Because we have not built the applicant show page yet, for now you can verify the file is attached in the Rails console. Start the Rails console with `rails c` in your terminal and then:

```
Applicant.last.resume.attached?
=> true
```

File uploading can be slow, especially for users on mobile networks. To improve performance and prevent our application servers from being kept busy uploading files and then transmitting them to a third party storage service, we can use the **direct upload** feature of `ActiveStorage`.

Direct uploads allow us to send a file directly from the browser to a cloud storage provider, completely bypassing our own application servers. Adding direct upload capability to a file field requires two modifications to our existing code.

The first step is to import and start `ActiveStorage's` JavaScript in `app/javascript/application.js`

app/javascript/application.js

```
19 import * as ActiveStorage from "@rails/activestorage"
20
21 ActiveStorage.start()
```

Step two is updating the `resume` file field to add the `direct_upload` attribute:

app/views/applicants/_form.html.erb

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

Easy.

Note that in development direct uploads are just regular uploads. In production, you will interface with a cloud storage provide like Amazon S3 for direct uploads. See the full list of supported cloud storage providers in the **Rails guides**.

In the next section, we will add the ability to move applicants between hiring stages by dragging and dropping the applicant's card.

Drag applicants between stages

Drag-and-drop is a very common feature in web applications. In this section we will take a look at two different methods for implementing drag-and-drop on the applicants page.

Users will drag-and-drop applicants between hiring stages in order to track the current hiring stage for each applicant. As an applicant progresses through the hiring process, users move them between the hiring stage columns on the Applicants page. In a commercial applicant tracking system, this Kanban board style layout can become very difficult to use as the number of applicants on the page increases. However, for small numbers of applicants this interface is intuitive and, despite the poor commercial viablity of this applicants page layout, it presents a good learning opportunity for us.

Each time an applicant is dragged to a new hiring stage, we will update the applicant's hiring stage in the database without making any changes to the UI.

In the first solution, Stimulus will be used to send a PATCH request to a server endpoint to update applicant hiring stages after a drag event occurs. In the second solution, we will use StimulusReflex to accomplish the same thing without a manually constructed PATCH request.

In both methods, we will allow users to drag applicants between hiring stage columns in the UI using **SortableJS**.

To begin, we need the SortableJS JavaScript package installed, so we will start there. From your terminal:

```
yarn add sortablejs
```

Dragging applicants with Stimulus

The Stimulus-only approach is an opportunity to examine another common use case for Stimulus controllers, integrating third party JavaScript libraries into your user interface. Let's see what this looks like.

Start by generating a new Stimulus controller. From your terminal:

```
rails g stimulus drag
```

Fill the new drag Stimulus controller:

app/javascript/controllers/drag_controller.js

```
import { Controller } from 'stimulus'
import Sortable from 'sortablejs'

export default class extends Controller {
  static targets = [ 'list' ]

  listTargetConnected() {

this.listTargets.forEach(this.initializeSortable.bind(thi
  }

  initializeSortable(target) {
    new Sortable(target, {
      group: 'hiring-stage',
      animation: 100,
      sort: false
    })
  }
}
```

In the controller, we declare a **`list`** target and then in **`listTargetConnected`** we loop through each **`list`** target in our controller and call **`initializeSortable`** on the **`list`** element. **`[name]TargetConnected`** is a built-in **Stimulus callback** that runs each time the target element is added to the DOM.

`initializeSortable` makes the target element a sortable list. The **group option** is how we enable dragging between different sortable lists on the same page. Note that the **`hiring-stage`** group name can be anything we like.

There are two key concepts to take note of in this controller. The first is that an instance of a Stimulus controller can have any number elements with the same **`target`** identifier. w=We can get all target elements at once with **`[targetName]Targets`**, making it trivial to act on multiple child elements at once in a controller.

The second is that Stimulus callbacks (like **listTargetConnected** and **controller lifecycle callbacks**) make it easy for us to add behavior from third party JavaScript libraries to our code.

Instead of listening to page-level events on the document or window, Stimulus utilizes mutation observers that allow us to attach behavior when an element enters or exits the DOM. This pattern of adding behavior on **`connect`** or **`initialize`** is one you will come back to regularly when working with Stimulus.

Now we will connect the controller to the DOM. In **`app/views/applicants/index.html.erb`**, update the applicant stages container div like this:

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: "mouseup->slideover#open", remote: true } %>
</div>
<div class="flex items-baseline justify-between">
  <div class="flex flex-grow mt-4 space-x-6 overflow-
auto" data-controller="drag">
    <% [:application, :interview, :offer, :hired].each
do |key| %>
      <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"><%=
key.to_s.humanize %></span>
        </div>
        <div id="applicants-<%= key %>" class="h-full"
data-drag-target="list">
          <% @applicants.where(stage: key).each do
|applicant| %>
            <%= render "card", applicant: applicant %>
          <% end %>
        </div>
      </div>
    <% end %>
  </div>
</div>
```

Here we added the `drag` controller to the `div` wrapping all of the applicant stage columns and added the `data-drag-target="list"` attribute to each column.

When the applicants index page loads, the `drag` controller `connect` method is called and each stage column becomes an individual Sortable list. Because each list has the same `group` attribute, we can drag applicants between lists.

We added `sort: false` in Sortable's options. This option prevents dragging applicants to a new position *within* the same group, you can only drag them to a new group. This is done intentionally, because we will not be allowing users to persist the order of applicants within a stage in this book.

You will notice that dragging between stages at this point "works", but only until the page is reloaded. We are not persisting stage changes in the database yet so applicant's move back to their original stage each time the page is reloaded. Let's tackle persistence next.

First, we will update the `drag` controller to send a PATCH request when the user finishes dragging an applicant, hooking into Sortable's `onEnd` option:

app/javascript/controllers/drag_controller.js

```
import { Controller } from 'stimulus'
import Sortable from 'sortablejs'

export default class extends Controller {
  static targets = [ 'list' ]
  static values = {
    url: String,
    attribute: String
  }

  connect() {

this.listTargets.forEach(this.initializeSortable.bind(thi
  }

  initializeSortable(target) {
    new Sortable(target, {
      group: 'shared',
      animation: 100,
      sort: false,
      onEnd: this.end.bind(this)
    })
  }

  end(event) {
    const id = event.item.dataset.id
    const url = this.urlValue.replace(":id", id)
    const formData = new FormData()
    formData.append(this.attributeValue,
event.to.dataset.newValue)
    window.mrujs.fetch(url, {
      method: 'PATCH',
      body: formData
    }).then(() => {}).catch((error) =>
console.error(error))
  }
}
```

Here, we are using **values** to define a `url` and an `attribute`. Using values enables us to define the URL and attribute that we want to update in the DOM, instead of hardcoding those values in the controller. By defining these values when we initialize the controller, we make the controller easier to reuse in other places in our application.

We also added a new `end` function which is called when Sortable's `onEnd` event is triggered. `end` grabs the id of the applicant that was moved and combines it with the `url` and `attribute` values to construct a PATCH request to the server.

We need to update the markup to define these values when we initialize the Stimulus controller before this controller will work as expected.

Back to the applicants index view, starting with the div with the `drag` controller attached:

app/views/applicants/index.html.erb

```
<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]">
  <% [:application, :interview, :offer, :hired].each do
|key| %>
    <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"><%=
key.to_s.humanize %></span>
      </div>
      <div id="applicants-<%= key %>" data-drag-
target="list" data-new-value="<%= key.to_s %>"
class="h-full">
        <% @applicants.where(stage: key).each do
|applicant| %>
          <%= render "card", applicant: applicant %>
        <% end %>
      </div>
```

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

Here, we updated the controller element to define the `url` and `attribute` value, and we updated the `list` target element to add the column's stage as a `new-value` data attribute.

We need each applicant to set a `data-id` attribute to use when we construct the final url for the PATCH request. Make that change in the `card` partial:

app/views/applicants/_card.html.erb

```
<div class="flex flex-col pb-2 overflow-auto" data-id="
<%= applicant.id %>">
  <!-- Snip -->
</div>
```

The last step to persisting stage changes in the database is to add a route and controller action to handle the stage change PATCH request.

We will do this by adding a non-RESTful `change_stage` route to the Applicants resource.

If we want to keep the entire application RESTful, we could define a new `Applicants::StageChangesController` with an `update` action. However, that would be a lot of extra ceremony for very little extra learning benefit, so we are going to break the rules a bit by using a non-RESTful action. Don't tell DHH about this, please.

Update the application's routes to add the new `change_stage` route:

config/routes.rb

```
resources :applicants do
  patch :change_stage, on: :member
end
```

Now define the `change_stage` action in the `ApplicantsController`:

app/controllers/applicants_controller.rb

```
before_action :set_applicant, only: %i[ show edit
update destroy change_stage ]

def change_stage
  @applicant.update(applicant_params)
  head :ok
end
```

Refresh the applicants page after adding the new route and action. Move applicants between stages and then refresh the page to see that your changes are now saved to the database.

Now that we have seen a Stimulus-powered drag-and-drop interface, let's reset and look at the same interface built with StimulusReflex. Like in the last chapter, we will implement two different solutions to the same problem, comparing and contrasting so you can get comfortable with the different options available to you as a Rails developer.

Sidebar: Drag-and-drop with StimulusReflex

The path forward with StimulusReflex-powered drag-and-drop is similar to the Stimulus powered approach. We will have a Stimulus controller that handles initializing Sortable on draggable elements. The Stimulus controller will hand things off to the backend when the Sortable `end` event is triggered, and the backend will silently save the applicant's new stage in the database.

To get started, generate a new **reflex** using the generator provided by StimulusReflex:

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

This generator creates a client-side Stimulus controller (`draggable_controller.js`) and a server-side reflex (`draggable_reflex.rb`) that work together to produce the desired end user experience.

Because new StimulusReflex controllers do not automatically update the Stimulus manifest file, whenever we generate a new Stimulus controller with StimulusReflex, we must follow that command with `rails stimulus:manifest:update`. The `manifest:update` task scans the `app/javascript/controllers` directory and registers each controller in the `controllers` directory in the Stimulus manifest file found at `app/javascript/controllers/index.js`.

StimulusReflex-enabled Stimulus controllers are very similar to regular Stimulus controllers, with the added bonus of being able to **trigger server-side Ruby code** effortlessly.

To see this in action, fill in the `draggable` Stimulus controller like this:

app/javascript/controllers/draggable_controller.js

```
import ApplicationController from
'./application_controller'
import Sortable from 'sortablejs'

export default class extends ApplicationController {
  static targets = [ 'list' ]
  static values = {
    attribute: String,
    resource: String
  }

  connect () {
    super.connect()

this.listTargets.forEach(this.initializeSortable.bind(thi
  }

  initializeSortable(target) {
    new Sortable(target, {
```

```
      group: 'shared',
      animation: 100,
      sort: false,
      onEnd: this.end.bind(this)
    })
  }

  end(event) {
    const value = event.to.dataset.newValue
    this.stimulate(
      "Draggable#update_record",
      event.item,
      this.resourceValue,
      this.attributeValue,
      value
    )
  }
}
```

This code should look pretty familiar. Let's walk through what is different.

First, instead of importing `stimulus`, we import `application_controller` and extend that controller to enable StimulusReflex functionality in the controller.

The `super.connect()` in the `connect` lifecycle method is also part of the basic creation of a StimulusReflex controller.

The `end` function is where things get interesting. In the Stimulus-only version of this controller, `end` is responsible for constructing form data from the DOM event's data, building a URL, and sending a PATCH request to that URL.

In this version, the `end` function calls `this.stimulate` to **trigger a reflex action** on the server. In the stimulate call, we pass in data from the DOM event and values from the Stimulus controller. These arguments are used by the server-side reflex action to find and update the correct applicant in the

We will define the `update_record` action in the reflex class next. Head to the `DraggableReflex` and fill it in with:

app/reflexes/draggable_reflex.rb

```
class DraggableReflex < ApplicationReflex
  def update_record(resource, field, value)
    id = element.dataset.id
    resource = resource.constantize.find(id)
    resource.update("#{field}": value)

    morph :nothing
  end
end
```

This is mostly standard Rails code. `update_record` finds a resource by id and then updates that resource with the value passed to `update_record`. There are two important StimulusReflex concepts demonstrated here.

You will notice that the `this.stimulate` call in the Stimulus controller had 4 arguments (plus the reflex action name) but `update_record` only accepts 3 arguments.

This is because the second argument passed to `stimulate` is the `element` we are interested in. By default, `element` is the DOM element the Stimulus controller is attached to. In our case, we need access to the `id` attribute on the individual applicant being moved, so we override the `element` in the stimulate call. This allows the reflex on the server-side to access that element's data attributes.

The second concept to review is the unfamiliar `morph :nothing` call. Morphs are how we tell StimulusReflex what to update in the DOM.

By default, after a reflex action (like our `update_record` action) runs, StimulusReflex updates the entire body of the page by reprocessing the current controller action, sending the new HTML body to the frontend, and then making updates with morphdom.

In many cases, these automatic, full page updates are exactly what you need. No more thinking about client-side state, just let StimulusReflex update the page as efficiently as possible after state has changed on the server as a result of a reflex action.

In our case, we do not need any page updates after this reflex runs. The DOM is already up to date because the user dragged the applicant where they wanted them to be. The reflex action's only job is updating the database. Reprocessing a controller action and sending an HTML payload back to the frontend would be a waste of energy in this case.

This is when a **nothing morph** comes in handy. Nothing morphs tell StimulusReflex to do... nothing. Once the code in the reflex runs, StimulusReflex tells the client **the reflex finished** and then wraps up for the day, no reprocessing controller actions or rendering wasted HTML.

With that brief introduction to a few StimulusReflex concepts, let's wrap up this sidebar by updating the DOM to connect the `draggable` controller so that dragging and dropping applicants uses StimulusReflex.

Update the applicants index page:

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: "mouseup->slideover#open", remote: true } %>
</div>
<div class="flex items-baseline justify-between">
  <div
    data-controller="draggable"
    data-draggable-resource-value="Applicant"
    data-draggable-attribute-value="stage"
```

```
    class="flex flex-grow mt-4 space-x-6 overflow-auto"
  >
    <% [:application, :interview, :offer, :hired].each
do |key| %>
      <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"><%=
key.to_s.humanize %></span>
        </div>
        <div id="applicants-<%= key %>" data-draggable-
target="list" data-new-value="<%= key.to_s %>"
class="h-full">
          <% @applicants.where(stage: key).each do
|applicant| %>
            <%= render "card", applicant: applicant %>
          <% end %>
        </div>
      </div>
    <% end %>
  </div>
</div>
```

Here we connect the controller using `**data-controller**`, add the `**data-draggable-value**` attributes, and then add the `**data-draggable-target="list"**` to each of the stage columns. All of these changes are very close to the Stimulus-only version of this feature

The card partial in this version of the feature is identical to the Stimulus-only version:

app/views/applicants/_card.html.erb

```
<div class="flex flex-col pb-2 overflow-auto" data-id="
<%= applicant.id %>">
  <!-- Snip -->
</div>
```

After the markup is updated, restart your server and then refresh the page and then drag applicants between stages. If all is well, the applicant stage changes should persist in the database as before. If you check the Rails server logs, you will see the nice, neat output from StimulusReflex broadcasting after each reflex action:

```
[4b77bdd6] 1/1 DraggableReflex#update_record -> document via
Nothing Morph (dispatch_event)
StimulusReflex::Channel transmitting {"cableReady"=>true,
"operations"=>[{"name"=>"stimulus-reflex:morph-nothing",
"selector"=>nil, "payload"=>{}, "stimulusReflex"=>{"attrs"=>
{"class"=>"flex flex-col pb-2 overflow-auto", "data-
id"=>"92c88316-8e05-476d-ab91-b7ff08ec8826"... (via streamed
from StimulusReflex::Channel:)
```

Now that we have seen drag-and-drop implemented with Stimulus and with StimulusReflex, moving forward this book will assume you have followed the Stimulus-only path. If you are use the StimulusReflex-powered version instead, be careful when copy and pasting updates to the applicants index page.

I chose to use the Stimulus-only case as the base for the rest of the book because StimulusReflex is probably a bit more than you need for a feature like this. If your application's UX never gets more complicated than this type of interaction, you will likely be best served by skipping the extra complexity that StimulusReflex introduces.

Later on in this book, we will take a look at more complex scenarios where StimulusReflex is the best tool for the job.

We have reached the end of chapter four, great work! In this chapter, we sharpened our CableReady skills with another slideover drawer implementation and then built a drag-and-drop interface with Stimulus. We also got our first taste of StimulusReflex by rebuilding the drag-and-drop interface with StimulusReflex as a sidebar. Before moving on to chapter five, pause and review any code that is not completely clear, commit your code if you are coding along with me, and then take a break if you need it.

When you are ready, move on to chapter five where we will add the ability to search and filter the applicants index page AND the jobs index page. It is going to get pretty wild.

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

Changelog

Change	Date	PR link
Updated `DragController` to use target connected callback instead of relying on the controller `connect` callback. This fixes a bug that introduced when filtering is added to the applicants page in Chapter 5. `connect` only runs once in our markup, while `listTargetConnected` runs each time the list is updated by the filtering feature.	March 1, 2022	PR 17 on Github