Extracted from:

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Covers Rails 7

Modern Front-End Development for Rails

Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

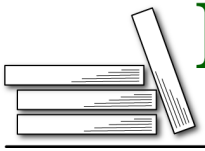
Edited by Katharine Dvorak

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-961-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2022

Communicating with the server is an important task for client-side apps. The server is usually the source of data truth, and it has information the client needs. When server information changes, we may need to send updated data to draw a new part of the client application. The server also often needs to be informed of actions taken on the client in order to update shared information.

In a Hotwire environment, you can use Turbo Frames and Turbo Streams to manage a lot of server interactivity through regular HTTP requests to the server, which then returns regular HTML responses. However, sometimes you may want to manage server communication as part of your Stimulus and React code.

In this chapter, we'll look at using Stimulus to mediate a form submission in our sample application to perform searches. Then, we'll discuss what to do when you have to contact a server that returns data rather than HTML. We'll also look at how to get React components to receive API data and incorporate it into their state by sending and receiving data about which seats in the concert have already been held.

Using Stimulus to Manage Forms

Turbo is designed to allow regular form submissions to trigger interactive changes on a page without having to reload the entire page. Sometimes, though, you need a little more client-side zest. You might want the form to be submitted on a user action other than clicking a Submit button. Or you might want to gather data from elsewhere on the page. In either case, you can use Stimulus to mediate your form submission.

If you look at the schedule page of our application, it has a search bar that currently does nothing except maybe offer typing practice. What we'd like it to do is return search results on typing. Specifically we want the following functionality:

- Typing in the box triggers a form submission with search results.
- Receiving search results uses Turbo Frames to place those results in a modal that overlays our page.
- Clearing the search field clears the modal.
- Clicking outside the modal window clears the modal.

For those of you who use the Hey email app,[1] this functionality is based on Hey's search functionality, though this is a simplified version of it.

To make this work, we need a polyfill. Generically, a *polyfill* is code that allows you to use a feature even if that feature has not been implemented on all browsers. Typically, the polyfill checks to see if the feature exists, and if it doesn't, the polyfill implements the feature (or a stripped-down version of it) on the browser so that code can be more easily shared between different browsers.

There's a slight difference in browser APIs as I write this, in that Apple's Safari browser does not implement a method we need to handle form submissions correctly. To rectify that, we need to use the polyfill package.

These days, Turbo includes this package by default, but if you are having trouble on Safari, you can install it yourself with the following command:

```
$ yarn add form-request-submit-polyfill
```

The method we need is called requestSubmit. (I'll explain why in a moment.)

Now, some logistics. To get client-side search to work, we need a functional server-side search. This isn't a book about server-side search optimization, so we'll go with something simple:

chapter_08/01/app/models/concert.rb

```
def self.search(query)
  joins(:bands)
    .where("concerts.name ILIKE ?", "%#{query}%")
    .or(Concert.where("concerts.genre_tags ILIKE ? ", "%#{query}%"))
    .or(Concert.joins(:bands).where("bands.name ILIKE ?", "%#{query}%"))
    .uniq
end
```

In this code, we're using SQL like statements to return any concert where the concert name, a band's name, or the list of genres contains the query string as a substring. We're throwing a uniq on the end because otherwise we'll get duplicate entries if multiple band names match. This is in no way complex enough to be a full production-level search, but it's fine for now.

We need to create a regular Rails endpoint for this search, and the natural place for it is the index method of the ConcertsController:

chapter_08/01/app/controllers/concerts_controller.rb

```
def index
  @query = params[:query]
```

1. http://hey.com

```
  @concerts = Concert.search(@query)
end
```

We also need a view for this. Here's the outer part:

chapter_08/01/app/views/concerts/index.html.erb

```
<%= turbo_frame_tag("search-results") do %>
  <article
      class="fixed bg-gray-300 z-10
             rounded-3xl ring-4 ring-gray-800
             max-w-screen-lg w-full
             mr-20 ml-32 px-6 py-2 mt-2
             overflow-y-auto overscroll-contain"
      data-search-target="results">
    <div class="text-3xl font-bold text-center">Search Results</div>
    <%= render(
          partial: "concerts/search_result",
          collection: @concerts,
          as: :concert,
          locals: {query: @query}
        ) %>
  </article>
<% end %>
<%= link_to("New Concert", new_concert_path) %>
```

This is a reasonably standard Rails view. It is surrounded by a Turbo Frame called search-results (aptly named because it's going to contain search results). We've also got a little Stimulus nugget in there, data-search-target="results", which will make more sense in a moment. And we've got an outer article tag with a big pile of Tailwind CSS that translates, roughly, to: "Put this element in a fixed position with a z index so that it will display over the underlying display like a modal, and give it a light gray background color, rounded corners, and an outline ring. Position it appropriately (that's the third and fourth line starting with max-w-screen and ending with mt-2), and make sure overflow text is scrollable." The Tailwind is a little more succinct. Effectively we get some HTML that will look like a modal box on top of our existing schedule.

In addition, we also have an internal partial named concerts/search_result that renders each individual concert, using the standard Rails shortcut of passing the entire collection to the collection option of the render method. I used a separate partial here rather than concerts/concert to allow for slight difference in how a search result is displayed versus a regular concert listing. For example the search result lists date and time, where the regular concert listing only lists time, since regular listings in this app are grouped by date. There's not much logic to results partial, but for the sake of completeness, here it is:

chapter_08/01/app/views/concerts/_search_result.html.erb

```
<article class="my-6 max-w-screen-lg">
  <div>
    <%= concert.start_time.by_example("Jan 2 @3:04 PM") %>
    <div class="font-bold text-xl">
      <%= link_to(concert, data: {"turbo-frame": "_top"}) do %>
        <%= highlight(concert.name, /#{query}/i) %>
      <% end %>
      <div class="float-right">
        <% if concert.sold_out? %>
          Sold out
        <% else %>
          <%= pluralize(concert.unsold_ticket_count, "Tickets") %>
          Remaining
        <% end %>
      </div>
    </div>
    <div>
      <%= highlight(concert.bands.map(&:name).join(", "), /#{query}/i) %>
    </div>
    <div><%= concert.genre_tags.split(",").to_sentence %></div>
    <div><%= concert.venue.name %></div>
  </div>
</article>
```

I want to point out here that we have a functional, if not fancy, Rails search. If you go to localhost:3000/concerts, you'll see an index page. If you go to localhost:3000/concerts?query=rock, you'll see search results. They'll look a little odd because we've styled them for a modal display, but they're not that bad. My point is that we can think about this as a progressive enhancement. We might have started with search being on a completely different page, but we've decided to make it part of the existing page for a better user experience.

When dealing with Hotwire functionality, it is almost always a good idea to get a server-side only version of the functionality working before starting to add client-side interaction. Doing the work this way will often result in a simpler final design, and you may find ways to do things in plain Turbo that you thought would require custom JavaScript.

Let's make that client-side interaction and modal work. First, we need to put a working form into the schedule display page. As currently written, we have a text_field_tag on that page that is surrounded by a couple of divs. That's our form. I'm going to pull out that entire div, currently, the one that is class="flex justify-center" and put it in its own partial:

chapter_08/01/app/views/schedules/_search_form.html.erb

```
<%= turbo_frame_tag("search-form") do %>
  <div data-controller="search">
```

```
    <%= form_with(
          url: concerts_url,
          method: "get",
          data: {
            "turbo-frame": "search-results",
            "search-target": "form",
            action: "input->search#submit"
          }
        ) do %>
      <div class="flex justify-center">
        <div class="w-4/5">
          <%= text_field_tag(
                "query", "",
                placeholder: "Search concerts",
                type: "search",
                id: "search_query",
                "data-search-target": "input",
                class: "w-full px-3 py-2
                        border border-gray-400 rounded-lg"
              ) %>
        </div>
      </div>
    <% end %>
    <%= turbo_frame_tag("search-results") %>
  </div>
<% end %>
```

A few things are going on here.

We've surrounded the whole thing with a Turbo Frame named search-form, which is there solely to ensure that the form submit is inside a Turbo Frame so that its submission will be considered a Turbo request. We've also got a div with a data-controller named search that declares the Stimulus controller we're going to write.

The actual text field is now surrounded by a normal Rails form_with form that submits a GET request to concerts_url, which Rails will interpret as ConcertsController with an index action—exactly the controller action for search that we just wrote.

The form has three data attributes, all of which are of interest to the Hotwire world. We declare data-turbo-frame as search-results, meaning that when the form is submitted, the response HTML should have a search-results frame that replaces the existing search-results frame, which you can see is at the end of the partial. We declare data-search-target, marking this element as the form target of the search controller. And we declare a Stimulus data-action so that when the form receives an input event, the submit method of the search controller is invoked.

The text field itself only changes to declare itself the input target of the search controller, and as mentioned, we end the snippet with an empty Turbo Frame named search-results.

Now all we need is the Stimulus controller to manage this. We haven't put a Submit button on the page—instead, we want the form to submit and display results whenever anything is typed in the text field.

Our first pass at the search controller is quite short—remember we need to run rails stimulus:manifest:update for this to be available.

chapter_08/01/app/javascript/controllers/search_controller.ts

```
import { Controller } from "@hotwired/stimulus"
import "form-request-submit-polyfill"

export default class SearchController extends Controller {
  static targets = ["form", "input"]
  formTarget: HTMLFormElement

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

What happens in this code right now is that any typing in the text field triggers a DOM input event, which then propagates up to the form. The form has declared a Stimulus action input->search#submit so when it receives the input event, the submit method of this controller is invoked. All that method does is requestSubmit on the form, which submits the form and does so with the event Turbo Frames is looking for. (The Safari method for submitting forms doesn't send the event Turbo needs; it doesn't have this method, which is why we have to import the polyfill.)

And this works. Run it, type in the text box, and you'll get search results. Thanks to the Rails highlight helper, you'll even get the matching part of each string highlighted.

What's happening here is that we are using JavaScript to submit a regular form request, and then Turbo Frames is parsing the regular result from that HTTP call. Because the form tag specifies data-turbo-frame=search-results, Turbo is parsing out the search-results Turbo Frame from the HTML response and inserting it in the page where we have already placed a blank Turbo Frame with the search-results ID.

A few niceties are still worth adding. Right now there's no way to get rid of the modal once it pops up. Also, the search can get tangled up if the user types too fast. We can fix both of those problems with a little more Stimulus.