Hotrails

← Back to the list of chapters

Turbo Drive

Published on January 23, 2022

In this chapter, we will explain what Turbo Drive is and how it speeds up our Ruby on Rails applications by converting all link clicks and form submissions into AJAX requests.

Understanding what Turbo Drive is

Turbo Drive is the **first part of Turbo**, which gets installed by default in Rails 7 applications, as we can see in our `Gemfile` and our JavaScript manifest file `application.js`:

```
# Gemfile

# Hotwire's SPA-like page accelerator [https://turbo.hotwired.dev]
gem "turbo-rails"
```

```
// app/javascript/application.js

// Entry point for the build script in your package.json
import "@hotwired/turbo-rails"
import "./controllers"
```

By default, Turbo Drive speeds up our Ruby on Rails applications by converting all link clicks and form submissions into AJAX requests. That means that ***our***

CRUD application from the first chapter is already a single-page application, and we had no custom code to write.

With Turbo Drive, our Ruby on Rails applications will be **fast by default** because the HTML page we first visit won't be completely refreshed. When Turbo Drive intercepts a link click or a form submission, the response to the AJAX request will only serve to replace the `<body>` of the HTML page. In *most cases*, the `<head>` of the current HTML page won't change, resulting in a considerable performance improvement: **the requests to download the fonts, CSS, and JavaScript files will only be made *once* when we first access the website**.

How does Turbo Drive work?

Turbo Drive works by intercepting "click" events on links and "submit" events on forms.

Every time a link is clicked, Turbo Drive intercepts the "click" event, overrides the default behavior by transforming the HTML request the link click would normally trigger into an AJAX request. When Turbo Drive receives the response, it replaces the `<body>` of the current page with the `<body>` of the response, leaving the `<head>` unchanged *in most cases*.

That's why new Ruby on Rails 7 applications are *single-page* applications by default. The page we first visit won't be completely replaced; only the `<body>` tag will.

For clicks on links, a pseudo-code implementation would look like this:

```
// Select all links on the page
const links = document.querySelectorAll("a");

// Add a "click" event listener on each link to intercept the click
// and override the default behavior
links.forEach((link) => {
  link.addEventListener("click", (event) => {
    // Override default behavior
    event.preventDefault()
    // Convert the click on the link into an AJAX request
    // Replace the current page's <body> with the <body> of the response
    // and leave the <head> unchanged
```

```
  }
)});
```

The same logic applies to form submissions. When a form is submitted, Turbo Drive intercepts the "submit" event and overrides the default behavior by transforming the form submission into an AJAX request and replacing the `<body>` of the current page with the `<body>` of the response, leaving the `<head>` unchanged.

For form submissions, a pseudo-code implementation would look like this:

```
// Select all forms on the page
const forms = document.querySelectorAll("form");

// Add a "submit" event listener on each form to intercept the submissio
// and override the default behavior
forms.forEach((form) => {
  form.addEventListener("submit", (event) => {
  // Override default behavior
    event.preventDefault()
    // Convert the form submission into an AJAX request
    // Replace the current page's <body> with the <body> of the response
    // and leave the <head> unchanged
  }
)});
```

As discussed in chapter 1, there is a breaking change in Rails 7: **Invalid form submissions have to return a 422 status code for Turbo Drive to replace the `<body>` of the page and display the form errors**. The alias for the 422 status code in Rails is `:unprocessable_entity`. That's why, since Ruby on Rails 7, the scaffold generator adds `status: :unprocessable_entity` to `#create` and `#update` actions when the resource couldn't be saved due to an invalid form submission.

Note: If you've been working with Rails for some time, you might be familiar with Turbolinks. Turbolinks is the ancestor of Turbo Drive: it only intercepted clicks on links but not form submissions. Now that Turbo also handles form submissions, the authors renamed the library from Turbolinks to Turbo Drive.

Disabling Turbo Drive

We may want to disable Turbo Drive for certain link clicks or form submissions in some cases. For example, this can be the case when working with gems that don't support Turbo Drive yet.

At the time writing this chapter, the Devise gem does not support Turbo Drive. A good workaround is to disable Turbo Drive on Devise forms such as the sign-in and sign-up forms. We will come back to this problem in a future chapter but for now, let's learn how to disable Turbo Drive on specific links and forms.

To disable Turbo Drive on a link or a form, we need to add the `data-turbo="false"` data attribute on it.

On the `Quotes#index` page, let's disable Turbo Drive on the "New quote" link:

```
<main class="container">
  <div class="header">
    <h1>Quotes</h1>
    <%= link_to "New quote",
                new_quote_path,
                class: "btn btn--primary",
                data: { turbo: false } %>
  </div>

  <%= render @quotes %>
</main>
```

Now let's refresh the page and click on the "New quote" link. We will notice the unpleasant white "blink" of the full HTML page refresh. If we perform this action again with the dev tools opened on the "Network" tab, we will notice that the browser makes four requests to the server:

- One **HTML** request to load the `Quotes#new` HTML page
- One request to load the CSS bundle
- One request to load the JavaScript bundle
- One request to load the favicon of the page

Note: I am using Google Chrome to perform this experiment. You might see slightly different results with other browsers.

Now let's add Turbo Drive back by removing the `data-turbo="false"` from the "New quote" link, refresh the page and perform the experiment again. We won't see the unpleasant white "blink" because the browser does not fully reload the page. Under the hood, Turbo Drive converts the click into an AJAX request and swaps the `<body>` of the page with the response's `<body>`. In the dev tools, we should see only two requests:

- One **AJAX** request to get the HTML for the `Quotes#new` page
- One request to load the favicon of the page

We can perform the same experiment on the form for quotes. Let's disable Turbo Drive on it:

```
<%= simple_form_for quote,
                    html: {
                      class: "quote form",
                      data: { turbo: false }
                    } do |f| %>

  <% if quote.errors.any? %>
    <div class="error-message">
      <%= quote.errors.full_messages.to_sentence.capitalize %>
    </div>
  <% end %>

  <%= f.input :name, input_html: { autofocus: true } %>
  <%= f.submit class: "btn btn--secondary" %>
<% end %>
```

Let's refresh our page and create a new quote with the dev tools opened on the "Network" tab. We should see the unpleasant white "blink" for the whole HTML page refresh and five requests:

- One **HTML** request to submit the form
- The **HTML** redirection to the `Quotes#index` page
- One request to load the CSS bundle
- One request to load the JavaScript bundle
- One request to load the favicon of the page

Let's remove the line we just added to re-enable Turbo Drive, refresh the page, and create another quote with the dev tools opened. We shouldn't see the unpleasant white "blink" and see only three requests:

- One **AJAX** request to submit the form
- The **AJAX** redirection to the `Quotes#index page`
- One request to load the favicon of the page

We demonstrated what Turbo Drive does for us in brand new Ruby on Rails 7 applications.

- It converts all link clicks and form submissions into AJAX requests to speed up our application
- It prevents the browser from making too many requests to load CSS and JavaScript files

The best part is that we didn't have to write any custom code. We get this benefit for free!

Note: It is also possible to disable Turbo Drive for the whole application, even though I don't recommend doing it as you will lose the speed benefits Turbo Drive provides.

To disable Turbo Drive on the whole application, we have to add two lines of config to our JavaScript code. You can, for example, do it directly in the manifest file:

```
// app/javascript/application.js

import { Turbo } from "@hotwired/turbo-rails"
Turbo.session.drive = false
```

Reloading the page with data-turbo-track="reload"

In *most cases*, Turbo Drive only replaces the `<body>` of the HTML page and leaves the `<head>` unchanged. I say in *most cases* because there are situations

where we want Turbo Drive to notice changes on the `<head>` of our web pages.

Let's take the example of a deployment where we change the CSS of our application. Thanks to the asset pipeline, the path to our CSS bundle will change from `/assets/application-oldfingerprint.css` to `/assets/application-newfingerprint.css`. However, if the `<head>` never changed, users that were on the website before the deployment and who remained on the website after the deployment would still be using the old CSS bundle as no request to download the new bundle would be sent. This could harm the user experience as users would use outdated CSS. We have the same problem with our JavaScript bundle.

To solve this problem, on every new request, Turbo Drive compares the DOM elements with `data-turbo-track="reload"` in the `<head>` of the current HTML page and the `<head>` of the response. If there are differences, Turbo Drive will reload the whole page.

If we have a look a the application layout of our app, we will notice that both assets tags were generated with the data attribute `data-turbo-track="reload"`:

```
<%# app/views/layouts/application.html.erb %>

<%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
<%= javascript_include_tag "application", "data-turbo-track": "reload",
```

Now that we know what the `data-turbo-track="reload"` data attribute does, a good exercise would be to demonstrate that it works as expected.

With the dev tools opened, let's click on a few links on our quote editor. We should see that the browser only sends AJAX requests to retrieve the HTML of the next page, as mentioned in the previous section.

Let's now make a silly temporary change to our CSS manifest to simulate a change in our CSS bundle and a deployment by, for example, importing the code for the `.btn` component twice:

```
// app/assets/stylesheets/application.sass.scss

// Remove the double import after the experiment
```

```
@import "components/btn";
@import "components/btn";
```

Next time we click on a link, we should see a complete page reload. Let's test it and see that it works! Turbo Drive is a fantastic piece of software!

Note: Small experiments like the one we just did help us understand what is happening in depth. We will do other experiments in the following chapters of this tutorial!

Changing the style of the Turbo Drive progress bar

As Turbo Drive overrides the browser's default behavior for link clicks and form submissions, the browser's default progress bar/loaders won't work as expected anymore.

Turbo has our back and has a built-in replacement for the browser's default progress bar, and we can style it to meet our application's design system! Let's style the Turbo Drive progress bar before moving to the next chapter:

```
// app/assets/stylesheets/components/_turbo_progress_bar.scss

.turbo-progress-bar {
  background: linear-gradient(to right, var(--color-primary), var(--colo
}
```

Let's not forget to import this Sass file into our manifest file:

```
// app/assets/stylesheets/application.sass.scss

// All the previous code
@import "components/turbo_progress_bar";
```

A good way to see we succeeded in adding styles to the Turbo progress bar is to **temporarily** add `sleep 3` to our controller actions for the progress bar to appear for at least 3 seconds:

```ruby
# app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  # Add this line to see the progress bar long enough
  # and remove it when it has the expected styles
  before_action -> { sleep 3 }
end
```

That's it, our progress bar is styled and matches our design system! We can now remove the `sleep 3` piece of code.

Conclusion

That was an easy chapter. We literally had nothing to do to make Turbo Drive work! We got substantial performance benefits for free without writing a single line of custom code!

In the next chapter, we will learn to transform complex pages into simple pieces using Turbo Frames. See you there!

← previous next →