Master Hotwire

Build Modern Web Apps with Rails Simplicity

(sample chapters)

Author: Radan Skorić

Table of Contents

0. Preface

Who this book is for and who it's not for

I consider your time precious, which is why I start here: so you can decide whether to continue reading.

What sets this book apart from other Hotwire books and courses is that it is written for **experienced Ruby on Rails developers**. This framing is the main differentiator. Most other books and courses assume a more junior reader. This focus allows me to give you a **deeper understanding** while demanding **less time** from you.

This book **IS** for:

- Experienced Ruby on Rails developers with no Hotwire experience.
- Developers who have used Hotwire but aren't yet comfortable with it.

This book **IS NOT** for:

- Developers with no Ruby on Rails experience.
- People inexperienced with software development.
- People who already have complete mastery of Hotwire and deep understanding of its internals.

The book **assumes a certain level of seniority** and doesn't cover introductory concepts, which means:

- If you're new to software development, this book is **not** for you.
- If you're an experienced Rails developer wanting to quickly learn Hotwire, this book is **perfect** for you.
- If you're experienced with a different backend framework, the book will still be **useful** for learning Hotwire, but you'll need to adapt the backend parts yourself.

How to use this book

The book has three parts, very imaginatively named **Part I**, **Part II**, and, you guessed it, **Part III**!

Ok, they also have subtitles, I'm not completely devoid of imagination.

Part I: Everything to finish the web application

This part introduces all Hotwire concepts in a practical manner through building a collaborative Kanban board application. You'll get the most value by developing alongside the book. There's an intentional order in the way the concepts are introduced.

Part II: How to package a web app into native mobile apps with Hotwire Native *(coming soon)*

This part guides you through packaging your web app into mobile apps for iOS and Android using Hotwire Native. You'll learn how Hotwire Native works and learn the basics of mobile development, just enough to allow you to launch a mobile app.

Part III: Take it further *(coming soon)*

This part contains additional chapters that either dive deeper into topics from Part I (and some from Part II) or introduce a new topic. The new topics are something you're likely to need after you put your Hotwire application into production. Unlike the first two parts, which should be read sequentially, these chapters can be read in any order. Each chapter is self-contained, with any dependencies clearly referenced or explained.

How chapters are structured

Chapters mostly follow this structure:

1. Introduction of a new concept in abstract terms.
2. Application of the concept to implement a specific feature on the example application.
3. Explanation of how the concept works under the hood.

While you could technically skip the "under the hood" sections, I **strongly** recommend reading them if time permits. This deeper understanding will prove invaluable when applying what you've learned. Combined with your existing experience, these insights will significantly accelerate your learning.

Why doesn't it include all of the Hotwire features?

Spoiler alert: It is not the intention of this book to cover every little feature of Hotwire. This is what the community maintained official documents are for.

The official documentation is great at helping you find a specific feature of Hotwire when you **are already experienced with it**. However, it is **not good** at giving you the high level understanding and a workable mental model for deep understanding of Hotwire. **This is where this book comes in**.

Being experienced with a technology means being comfortable implementing solutions and knowing how to find information quickly – not memorizing every feature. This book serves as **an accelerated learning tool** to help you become **an experienced Hotwire developer**.

Tip

For reference, these are the official documents we will be using throughout the book:

- Official Hotwire website: https://hotwired.dev/.
- Turbo Rails gem README: https://github.com/hotwired/turbo-rails.
- Turbo documentation: https://rubydoc.info/github/hotwired/turbo-rails.
- Rails documentation: https://api.rubyonrails.org/.

What we will build

We'll build a collaborative Kanban board application called **Hotboard** (short for "Hotwire Kanban Board"). Think of it as a simplified version of Trello with these features:

- Create and organize tickets on a multi-column board.
- Single-page application feel with seamless user interactions.

- Drag-and-drop functionality for moving tickets between columns.
- Real-time collaboration where users see each other's changes instantly.

The application will be minimalist yet fully functional – suitable for managing real work on actual projects. I've made the code MIT licensed and public to:

1. Enable easy sharing of code snippets with your colleagues
2. Allow you to reuse code in your projects

You can try the live application at hotboard.masterhotwire.com. Feel free to experiment with it – it's a demo, just keep it professional! :)

Important

Find the repository at: `https://github.com/radanskoric/hotboard/`

As you're implementing each chapter you're encouraged to experiment and play with the code. As you do that, you might drift away from the book's code.

To make it easy to align back, relevant chapters will start with a `git checkout` command. Simply use it on the accompanying repo to get to a clean starting position for following that chapter.

If you're considering implementing a different application, I recommend following along with the Kanban board for your **first pass**. This will let you focus on learning the concepts in a structured, efficient way. Once you're comfortable with the concepts, you can apply them to your own application.

1.0. Part I - Where you learn how to implement a Hotwire web application

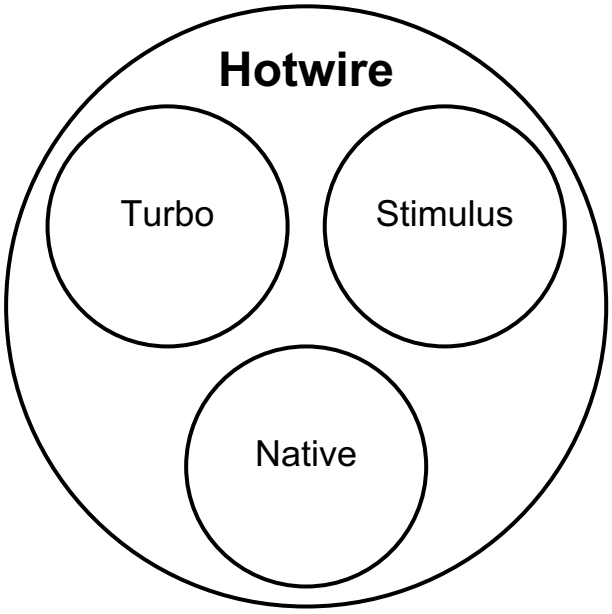
The learning starts here. We'll go through all aspects of implementing a Hotwire application from scratch. You will build a simple Kanban board. The implementation order is intentionally chosen to allow me to introduce various aspects of the Hotwire stack in a logical sequence and immediately apply them to the application we are building. Let's get started.

What is Hotwire?

At first, Hotwire can be a little bit confusing because there's no single "Hotwire" library. Instead, it's a collection of (mostly) JavaScript libraries that happen to fit nicely together. Hotwire is just an umbrella term for using them together.

There are three of them:

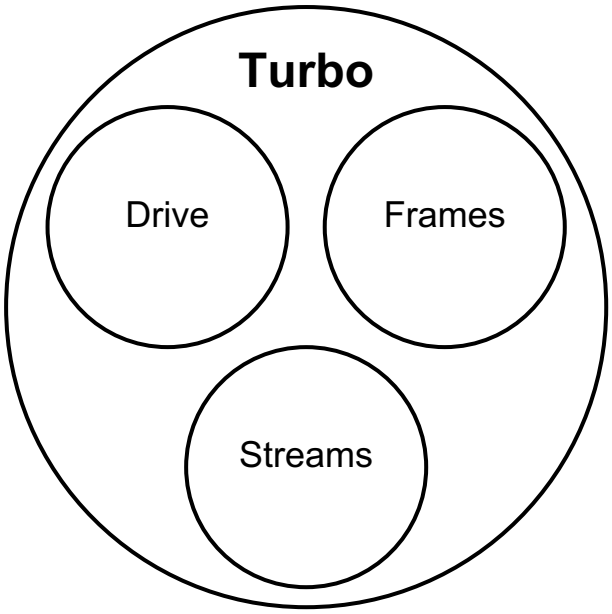
1. **Turbo** is responsible for enhancing browser capabilities and takes care of all server communication. It is probably the most important of all three, and we'll spend the most time with it.
2. **Stimulus** is a lightweight JavaScript UI library that makes it easy to add JavaScript-powered interactivity to your application. It also works really nicely with Turbo.
3. **Strada** is a library for taking your Turbo-powered web application and easily packaging it into a native application.



Each of these could theoretically be used on its own, but when they're brought together, they complement each other nicely. Together they form **Hotwire**. You will learn about all of them throughout this book.

Zooming in on Turbo, it contains three important concepts:

1. **Turbo Drive** implements the browser enhancements, the "magical" part.
2. **Turbo Frames** are the basic building blocks of a Turbo-powered app.
3. **Turbo Streams** give you finer control over the page content.



They are all part of the same JavaScript library, Turbo, but they are distinctly separate parts of the API.

If this is your first time hearing about these concepts, it might all seem a bit abstract and unclear. That's fine - you'll learn about each part in detail soon.

Let's get started.

1.1. Start with a plain HTML application

At the heart of Hotwire's compression of complexity is the simple idea that maybe we don't need to reinvent how browsers work from inside the browser itself. Browsers have had decades of engineering effort, optimizations, and new features added to HTML+CSS. Yes, there's JavaScript, but much more can be done in modern browsers without it. Hotwire asks: "What if we go back to server-side HTML+CSS and **enhance** it just enough to get a modern user experience?"

Enhance, rather than reinvent. That's the key insight.

This is also why, when you start to work on a Hotwire-powered application, it's best to start by thinking about **how a completely JavaScript-free version would look** Imagine if every action the user took was just a full-page interaction with the server. In other words: a plain HTML application where everything happens on the backend and there's no JS at all running in the browser. Let's start.

> **Important**
>
> I will be using Ruby on Rails for the backend, but Hotwire itself is completely backend-agnostic, and 90% of the content in this book will be applicable to any backend. That said, if you're comfortable with Rails, it will be a lot easier to follow along.

We'll first create a fresh new Ruby on Rails application. I'm assuming you're using at least Rails 8. It ships with ActionCable working out of the box with SQLite, which simplifies some of the steps. So please make sure that your Rails gem is up to date by running:

```Shell
gem install rails
```

Then create the application using the following options:

```Shell
rails new hotboard -c tailwind --skip-jbuilder
```

For styling, we'll use Tailwind. It's well integrated with Rails and has many resources and great documentation. You don't need to be familiar with it as I'll give you the right classes to use. It will allow us to not waste too much time with CSS. However, if you want to quickly get up to speed with it, I recommend watching the excellent RailsWorld 2023 talk by Adam Wathan, the creator of Tailwind, titled: "Tailwind CSS: It looks awful, and it works" (31 min).

Since we won't be implementing an API, we're skipping Jbuilder so it doesn't distract us. Keep it if you plan to add API functionality yourself.

Where is the Hotwire in the Rails application?

The Hotwire part is actually in an official gem that is included by default in every Rails application: `turbo-rails`. You could, for example, exclude it if you were building an API-only Rails application.

Most of the server-side helpers we'll be using are from this gem, and you won't find them if you go looking for them in the official Rails documentation. Instead, you'll have to look at the documentation for the turbo-rails gem.

Starting the Kanban board with Rails scaffolding

As we established, it's best to start with a plain HTML application. Fortunately for us, the Rails scaffold gives us exactly what we need. Out of the box, it creates a full interface where everything happens on separate pages, i.e., a plain HTML implementation.

We'll start simple by representing a ticket with just two attributes: title and description.

First, generate the scaffold:

```
Shell
bin/rails g scaffold Ticket title:string description:string
```

Then set up the database:

```
Shell
bin/rails db:create:all && bin/rails db:migrate
```

And link the root of the application to render tickets:

```
routes.rb
Rails.application.routes.draw do
  # ....

  # Defines the root path route ("/")
  root "tickets#index"
end
```

The scaffold generated tests in the `/test` folder. Now is a good time to verify that they work:

```
Shell
bin/rails test && bin/rails test:system
```

Now start the server so you can try the application manually:

```
Shell
bin/dev
```

This command will start the Rails server as well as the Tailwind processor. The application should be accessible at http://localhost:3000.

As we develop the application, I will not write about the tests. This is because there is nothing special about testing a Hotwire application. System tests can be written in the same manner you would write them for any UI implementation. And since the book is aimed at experienced developers, I am assuming you are familiar with the art of testing a Rails application.

However, the accompanying git repository implements every step of the application, and it also contains tests. If you're interested in testing, I advise reading the tests in the repository. They are written with the default Rails setup using Minitest. I personally prefer RSpec but opted to stick to

the defaults here to simplify the setup. If you'd prefer to use RSpec, you can use it as you code along.

1.2. Understand Turbo Drive enhancements

Over the next few chapters, we're going to take the plain HTML application and introduce Turbo to it. We'll start with Turbo Drive. Turbo Drive is by far the easiest part of Hotwire to use. In fact, we already have Turbo Drive running! The Rails generator includes it by default.

The role of Turbo Drive

So what is Turbo Drive?

To understand, let's first take a step back and consider what happens when you click a link to navigate to a new page within the same application:

1. The new page's HTML is loaded from the server.
2. The browser clears the memory, throws everything away, and starts from a clean state.
3. The browser parses the HTML, loading any assets it finds along the way. If they're in the browser cache, they're restored from the cache. Otherwise, the browser loads them from the server. However, wherever the CSS or JS assets came from, they get parsed and executed.

The browser has to do all that because it can't make assumptions about your website. But we know that we're navigating between pages of the same application. This also means that most assets, especially CSS and JS, don't change between page loads. The browser does a lot of redundant work that is not necessary. This is what Turbo Drive exploits to give us a speed boost.

What Turbo Drive does is:

1. It intercepts the link click or form submission and prevents the default browser action.
2. It performs a fetch request to load the full HTML, just like a browser would do.
3. However, unlike the browser, it then compares the current and new content of the `head` tag and processes each node separately: A. Nodes that are identical are untouched. B. Nodes that were removed in new content are removed from the page. C. New nodes are added into the page.
4. Only the `body` tag is replaced in whole with the new content.

The effect of all this is that unchanged assets between pages remain untouched, meaning they *don't need to be parsed and executed* This makes the browser's job significantly easier, giving us an almost free speed boost. The navigation becomes perceptibly smoother.

What Turbo Drive expects from our app to work correctly?

There is an important caveat to using Turbo Drive. Our application needs to conform to where it expects us to declare the assets, i.e., **in the `head` tag**. However, that is not a problem on modern browsers.

Some time ago, the best practice was to load CSS in the `head` but to load JS at the bottom of the `body` tag. This was done because the browser had to stop parsing the HTML when it was parsing JS. So loading JS in the `head` section would mean we would have to wait longer to see the HTML. This is no longer needed for several reasons:

1. Modern browsers are much more efficient at all of these tasks.
2. The `script` tag's `async` attribute is now widely supported. In short, it tells the browser

that it doesn't need to block on parsing this script. It can load and evaluate it in parallel with continuing to parse the HTML. You can read more about it in MDN docs.
3. HTTP/2 supports parallel download of multiple assets.

This means that today there are usually no reasons why we would still want to place script tags in the `body` tag.

Info

If you're adding Turbo to an existing application, you might not be able to comply with this requirement right away. You might be using JavaScript libraries which, by default, assume they are included in the `body` tag. In that case, it is possible to disable certain parts of Turbo that interfere with such libraries.

There will be a special chapter in the third part of the book dedicated to adding Hotwire and specifically Turbo to legacy applications. For now, rest assured that I am speaking from first-hand experience when I tell you that you can get tremendous value from Turbo even in a legacy application where you can't use all of its features.

Turbo Drive extra functionality (replacing UJS)

If you've been using Rails for a while, you've likely used the Unobtrusive JavaScript Adapter, also known as UJS. For quite a long time, it was the default Rails solution for a few common UI patterns. Turbo Drive covers all of the functionality UJS provided and more.

Taken directly from the official Rails guides:

> Rails 6 shipped with a tool called UJS (Unobtrusive JavaScript). UJS allows developers to override the HTTP request method of `<a>` tags, to add confirmation dialogs before executing an action, and more. UJS was the default before Rails 7, but it is now recommended to use Turbo instead.

Here we'll cover the 4 things that UJS provided and how to do it with Turbo Drive. These will be familiar to you if you've used UJS before, and as you'll see, they map cleanly to Turbo, except you'll also get all the other Turbo goodies you'll learn about.

Make non-GET requests from hyperlinks

Just like UJS used to, Turbo gives us the ability to do something not possible in plain HTML: Create links that end up sending a request other than GET, e.g., a POST or DELETE links.

Tip

Using this for DELETE links is a very common pattern. You might have a list of items and a little trash can icon next to each one. That trash can icon is most easily implemented as a DELETE method link.

Turbo implements this with the `data-turbo-method` attribute. This attribute can be used to mark links as executing a `POST` or `DELETE` call. I.e., you might use it like this to delete a ticket:

```ruby
link_to "Delete", ticket, data: {turbo_method: :delete}
```

… Next section is omitted from sample. If you're curious the following are the other 3 topics covered:

- *Force confirmation dialogs for various actions*
- *Make forms or hyperlinks submit data asynchronously with Fetch*
- Have submit buttons become automatically disabled on form submit to prevent double-clicking …

Using Turbo Drive programmatically

We've covered how you can use Turbo Drive through HTML. You can also use it directly from JavaScript.

Turbo exposes the visit function as part of its API, which can be called with a URL location:

```javascript
Turbo.visit(location)
```

Running this JavaScript will have the same effect as clicking an equivalent Turbo-enabled link: Turbo will navigate to the new location. This is the function you should be calling if you want to trigger Turbo navigation from your custom JavaScript code.

In many ways, this is **the** central function of Turbo, and we'll mention it often. It supports other use cases as well, but we'll cover those when we learn the relevant concepts.

How Turbo Drive works under the hood

When you load Turbo on the page, it defines all the necessary classes. It also initializes a few default objects internally, but at this point, they're all passive; they're not doing anything. Once it's all loaded, Turbo calls:

```javascript
Turbo.start()
```

This in turn just calls start on the session object, i.e., it's equivalent to:

```javascript
Turbo.session.start()
```

This is where the magic starts!

All of the enhancements that "automagically" make your page better without you doing anything are set up in that class. If you check the source of the Turbo Session class, you'll see that it's quite a list. We'll be returning back to this class when we're digging under the hood of other features, but for now, we'll look at just a few specific parts that we've already talked about at a higher level. Here's the session start function where I've removed the parts we're not yet interested in. This is so we don't get distracted:

hotwired/turbo/src/core/session.js

```
start() {
  if (!this.started) {
    // ...
    this.formLinkClickObserver.start()
    this.linkClickObserver.start()
    this.formSubmitObserver.start()
    // ...
    this.started = true
    this.enabled = true
  }
}
```

Tip

If you find yourself having to dig into Turbo to better understand why it's doing something it's doing (and it's beyond what I cover in this book), this is a great place to start. In general, you'll find almost all functionality is centred on the Session object as it's the one that holds all of the page-level state.

Let's unpack and explain the relevance of each of these and how they work!

Note

The source of Hotwire libraries generally doesn't contain comments as they're developed with the philosophy that code should be self-explanatory. Maybe that's me, but I like to have comments that explain **why** something exists as this is very hard if not impossible to encode into the code itself. I find an occasional high-level comment explaining **the purpose** of a class helps me a lot in understanding it.

To be fair, the code in all Hotwire libraries is of the highest quality, and I completely agree that there's no need for comments that explain **how** some code works. However, if you're like me, and are expecting some comments, don't be surprised when you dive into the source of these libraries.

How Turbo Drive Updates The Page

Let's for a moment set aside the question of how Turbo inserts itself to change the default browser behaviour and first look at how Turbo performs the page update once it has done the fetch request itself instead of the browser.

It handles the `body` and `head` sections of the page very differently.

For the **body**, the default action is to replace the complete content of the body with the new body. If you heard about "morphing" and are wondering where it fits in, rest assured that we'll dedicate a whole chapter to it. For now, we are sticking with the default method: **replace**.

For the **head** element, it does something different. It:

1. looks at each individual direct child of the current `head` element and checks if it is present in the new content. If not, it deletes it.

2. looks at each direct child of the new head element and if it is not present in the current one, it inserts it. Notice that these two put together mean that changed children are removed and a new version inserted.
3. doesn't do anything with children that are identical in both current and new content.

The important part of this is that elements which are the same in current and new `head` are not touched. This means that the browser doesn't have to parse and evaluate them again. And that right there is **most of the speed improvement** that Turbo Drive gives us. It makes a huge difference.

Note that order is ignored, so the final order of `head` children might end up different than it is in the payload. This is okay because in `head`, unlike in `body`, the order of elements doesn't matter, or at least, it shouldn't matter.

Note

Right now, you might be thinking: "Wait a minute! It matters for JavaScript! A later script might expect a library to be declared before it.". Yes, correct, but notice two things:

1. The tags are added in the same **relative** order as they appear in the new content.
2. Any other scripts are **already evaluated** and their relative order was already dictated by the previous page load.

In other words, any relative order of script tags that actually matters is preserved.

Now that we understand how page updating works, we can unpack the observers that each implement part of the Drive functionality. I'll explain them in a different order from how they are listed in the code, because they rely on each other and this order will make it easier to understand.

LinkClickObserver - the regular links

This is the observer responsible for Turbo intercepting link clicks and doing its Drive magic. This is usually the first Turbo feature people learn about: clicking links that would normally do a full page reload is *magically* faster because of Turbo Drive.

`LinkClickObserver` works by attaching a global listener to the `click` event on the `window` object. For any click that happens, it will:

1. Run a number of checks on the `event.target`, i.e., the element being clicked, designed to **prevent Turbo from intercepting clicks which are not navigation** (like clicking inside an input field).
2. **Locate the first `a` tag parent element.** This mirrors the browser behaviour: clicking inside an `a` tag bubbles up to it and makes the browser follow it.
3. Check if this link **can be followed**. This particular logic could cause issues for you in the corner cases, so I'll explain it in more detail below.
4. Finally, if the link can be followed, it will call `event.preventDefault` to stop the browser from navigating and instead call the `Turbo.visit` method. In other words, it calls the same `visit` method that is part of the public `Turbo` interface. It's useful to know because if you ever find yourself wanting to simulate a click on a link, know that you can

keep it simple and call `Turbo.visit` directly.
5. `Turbo.visit` will then update the page as we explained before.

Expanding on point 3, for the link to be **followable**, **ALL** of the following needs to be true:

1. **Turbo needs to be enabled on the particular link element.** This is not a trivial check because, in addition to being able to turn Turbo on and off globally, you can also selectively turn it on and off for an element and all its descendants by setting the `data-turbo` attribute to `true` or `false`. These can also be nested, and the innermost setting containing the link wins.
2. **The linked location (i.e., the `href` attribute) needs to be on your website.** Turbo will not take over navigation to a page outside your site because it has no way of knowing if that site would work with Turbo.
3. **Finally, it will give a chance to the app code to stop it from following the link.** It will dispatch a `turbo:click` event. If your application code catches that event and calls `preventDefault` on it, then Turbo will stop its own process and not intercept the link, letting the browser handle it as usual. This part may be a bit counterintuitive. To clarify: cancelling the `turbo:click` event will not prevent the page navigation. *It will just prevent Turbo from handling the navigation.* If you want to prevent the navigation itself, then prevent the `click` event itself.

FormSubmitObserver - the HTML forms

This is the observer that is responsible for form submissions working with the *Drive magic*.

> **Tip**
>
> If you're following along in the Turbo source code, note that the observer itself just takes care of catching form submissions and it delegates the actual submission handling logic to the `FormSubmission` class.

This part of Turbo does quite a lot; it wouldn't make sense to cover all of it even in this "deep dive" section. Instead, I'll highlight a few key things it does, roughly in the order they happen during the lifecycle of a form submission. My intention is to give you a good enough mental model so that you know where to look if you're running into issues with Turbo-powered form submissions.

So, in the order of execution, some highlights of what Turbo does on form submission:

1. It takes care of opening a confirmation dialog and waiting for the user to respond to it. Earlier in this chapter, you learned how you can customise this dialog by setting `Turbo.config.forms.confirm`.
2. It serialises all of the form fields using FormData native browser object so it can use it in a Fetch request.
3. Ensures that the "X-CSRF-Token" header value is correctly set. This is crucial for a Fetch-based submit to work with Rails. If you need to run a fetch request yourself, make sure to also set this header, or, alternatively, use Rails Request.JS library.
4. It marks the form submitter (e.g., the form submit button) as disabled while the form is submitting. This prevents accidental double clicks causing double submission.
5. It takes care of emitting `turbo:submit-start` and `turbo:submit-end` events. These

are very handy if you want to change the UI while the form is submitting, for example, by showing a loader on the submit button.
6. Once the response from the form submission arrives, it does extra processing. Specifically, if the response `Content-Type` is a turbo stream (i.e., `text/vnd.turbo-stream.html`), it handles processing of turbo streams. We'll learn about those in a few chapters; for now, just be aware they are automatically handled here.

All in all, quite a lot. There is also one specific piece of logic that almost everyone runs into: a **successful** form submission is expected to issue a redirect. If not, Turbo throws an error. This is problematic if the form has errors and you want to render the form back to show errors. The fix is simple. Turbo expects a redirect only for the successful responses. If the response status is in the 4xx range, Turbo will happily render its content. Typically, in the Rails controller, you will use `:unprocessable_entity`:

Ruby

```ruby
if @ticket.save
  # success logic
else
  render :new, status: :unprocessable_entity
end
```

This will be especially handy when we learn about Turbo Frames, and we'll mention it again there. You'll then already know that the `FormSubmission` object is responsible for this.

FormLinkClickObserver - the links that are not GET links

This observer is responsible for enabling `data-turbo-method` links that don't perform a regular GET request.

Under the hood, it uses the `LinkClickObserver`, which we already covered. It uses it to observe link clicks but scopes it down to links that:

1. Have a `data-turbo-method` attribute. OR
2. Have a `data-turbo-stream` attribute. This attribute marks a link as being able to receive a turbo stream response. We'll learn more about turbo streams later. For now, just be aware that they require special handling to work.

Since it's using `LinkClickObserver`, it will perform all of the checks we described in the previous chapter. The only difference is that in the end, instead of calling `Turbo.visit`, it will handle following the link itself differently.

At a high level, it creates a matching hidden form that it immediately submits. Specifically, it will:

1. Construct a hidden HTML form by directly constructing the HTML elements in memory.
2. Transfer all of the relevant attributes from the link to the new form. This includes the URL but also things like the submit method and whether there is a confirmation dialog set.
3. It will insert the hidden form into the document body.
4. Attach a listener that will remove the form when its submission completes.
5. Finally, it will submit the form.

This mirrors the way you would mimic POST and DELETE links with plain HTML, by disguising a form as a link. Turbo just does it on the fly.

And since it does it through the HTML, it's indirectly triggering the `FormSubmitObserver`,

which we just covered. Now you know that the special link handling relies on both the logic for regular links **and** for forms. Everything that applies to them also applies to the special links.

1.3. Enjoy the magic of link prefetching

This chapter is excluded from the sample.

1.4. Upgrade with Turbo Frames

Let's recap what we've built so far:

- We have a working Rails application.
- We have full CRUD operations for a global collection of tickets.
- It uses Turbo Drive to load pages faster, but all CRUD operations are on separate pages.

We got all of this almost for free from Rails generators for a new application and scaffolding.

This works well, but it's definitely far from being a modern web application. Remember what we said about the best way to build a new Turbo-powered application:

1. Start with a plain HTML application where every action is a separate page with full page load.
2. Take the pieces of that UI and compose separate pages into an integrated UI using Turbo.

You've already completed step one, the easy part - well done! And pretty soon you'll tackle step two!

What are Turbo Frames?

At the most basic level, Turbo Frames are custom HTML tags defined by Turbo. When placed around a part of the page, they capture any links and forms inside the frame and treat them as actions that update that frame:

```Html
Surrounding HTML content.

<turbo-frame id="foobar">
  Links and forms inside the frame are treated as frame updates:
  <a href="/more">I will update the frame, not the full page</a>
</turbo-frame>

More HTML content.
```

In a nutshell that is it. There's of course a lot more to them. We will learn it all by using them to add functionality to our Hotboard application.

Inline SPA-style editing with no JavaScript

We'll start by implementing inline editing of tickets, single-page application style.

Open the list of tickets in your browser. Right now, to edit a Ticket we need to:

1. Open the dedicated ticket page.
2. Click the "Edit" button which will take us to a new page with the form for editing the ticket.
3. **Make our changes and save the ticket.**
4. We're then redirected back to the page showing the ticket.
5. From there we can click a button to go back to the list of tickets.

The only useful part of that whole flow is the bolded 3rd step where we actually do the work. We want to do this without leaving the list of tickets - that is, we want **inline editing**.

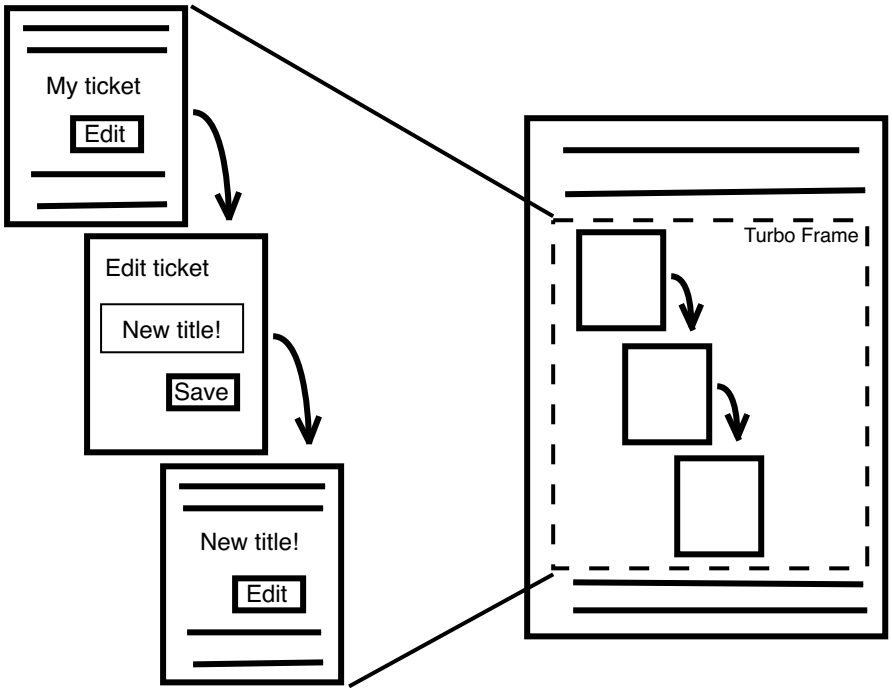
We will do that with Turbo Frames! Here's what we will do, from a high level:

1. "Frame" a section of the page around a single ticket. Or rather, we will Turbo "Frame" it. :D No, I am not ashamed of that pun.
2. We will take the above-described flow and push it all into the frame - that is, make it happen inside a part of the current page instead of in the browser window.

This will have two important effects:

1. Steps 1 and 5 are not needed since we're never leaving the current page.
2. Steps 2 and 4 become invisible since they're happening inside the frame.

This leaves just step 3, the part we actually want. Simple. Beautiful.



*Part of the website inside a section of the page*is a very useful mental model for Turbo Frames.

Implementation

Hotboard

Start the chapter from Hotboard repo with: `git checkout set-root-route`

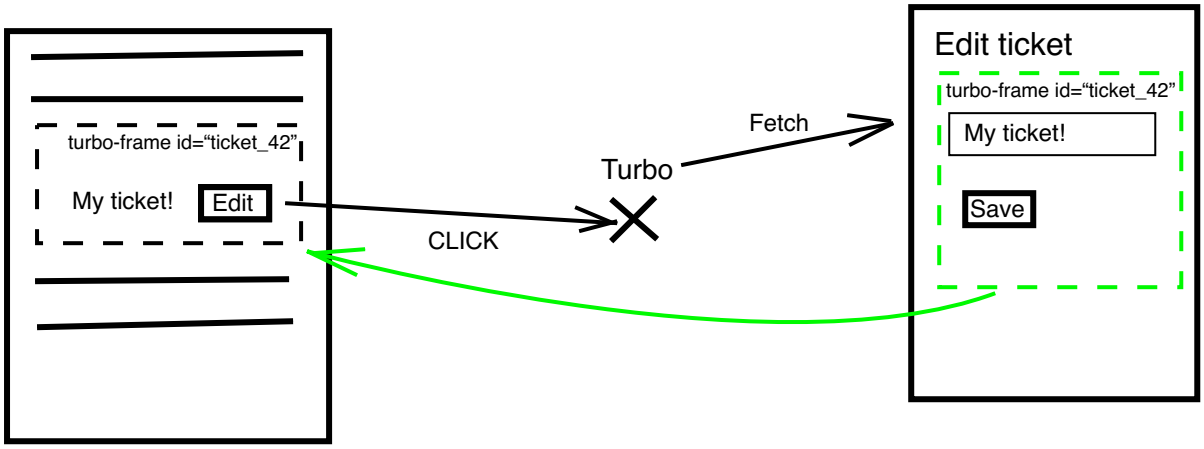
First, we need to frame the section of the index page where the flow will happen. We'll accomplish that by putting a frame around the `_ticket` partial, where we render a ticket. Also, to maintain a unique `id`, we'll add a "wrapper" prefix to the `div` already using the ticket id:

app/views/tickets/_ticket.html.erb

```
<%= turbo_frame_tag ticket do %>
  <div id="<%= dom_id ticket, :wrapper %>">
  ... old content of the partial ...
  </div>
<% end %>
```

Now, whenever we click a link inside a frame, Turbo will:

1. Find the first frame containing the link.
2. Intercept default browser behaviour and load the link itself by executing a `Fetch`.
3. Look for the frame tag with a **matching `id` attribute** in the **response content**.
4. Once it finds it, it will take the new frame content and replace the current frame content with it.



Turbo is inserting itself into the normal browser flow.

Notice that the response must contain a matching frame and we haven't added it yet! We'll do that now for the edit ticket view, just wrap the form rendering and the "show" link in the exact same frame tag:

app/views/tickets/edit.html.erb

```
...
<%= turbo_frame_tag @ticket do %>
  <%= render "form", ticket: @ticket %>
  <%= link_to "Show this ticket", @ticket, class: "..." %>
<% end %>
...
```

It's intentional that we're not framing the entire edit template in a frame. We only need a part of it for inline editing. The rest of the page is there if user opens the edit link as a full page.

We're still missing the "Edit" link when rendering the ticket in the list. The neat part is that, because of the above described Turbo logic, it can be a regular link. Simply **cut** the "Edit this ticket link" from `app/views/tickets/show.html.erb` and **paste** it into `_ticket.html.erb` partial with just one small change: rename `@ticket` to `ticket`:

app/views/tickets/_ticket.html.erb

```
<%= turbo_frame_tag ticket do %>
  ... old content ...
  <%= link_to "Edit this ticket", edit_ticket_path(ticket), class: "..." %>
<% end %>
```

A small note: the "Show this ticket" button that the scaffold put on the index page, right after `<%= render ticket %>`, is no longer needed, so go ahead and remove it.

Now refresh the page and click the edit link: the form will load directly in the page. Change the details of the ticket, click save and it will work. It was that easy.

Turbo doesn't only intercept link clicks, it does the same for form submissions. When you clicked save this is what happened:

1. Turbo intercepted the form submission, it serialised the form itself and executed the form submission via a Fetch call to the server, making sure to use the appropriate method (POST or GET).
2. The server responded with a redirect to the show action. The browser took care of following that redirect.
3. The show action loaded and since the show action uses the `_ticket` partial, it contained the relevant turbo frame.
4. Turbo located the frame by its id in the response content and updated the content of the frame on the page.

Allow deleting the tickets from the index page

. . .

This is the end of sample for Master Hotwire e-book (https://masterhotwire.com/).