Remember Mac and Me? (No?)

But how about ...

Boulder Ruby

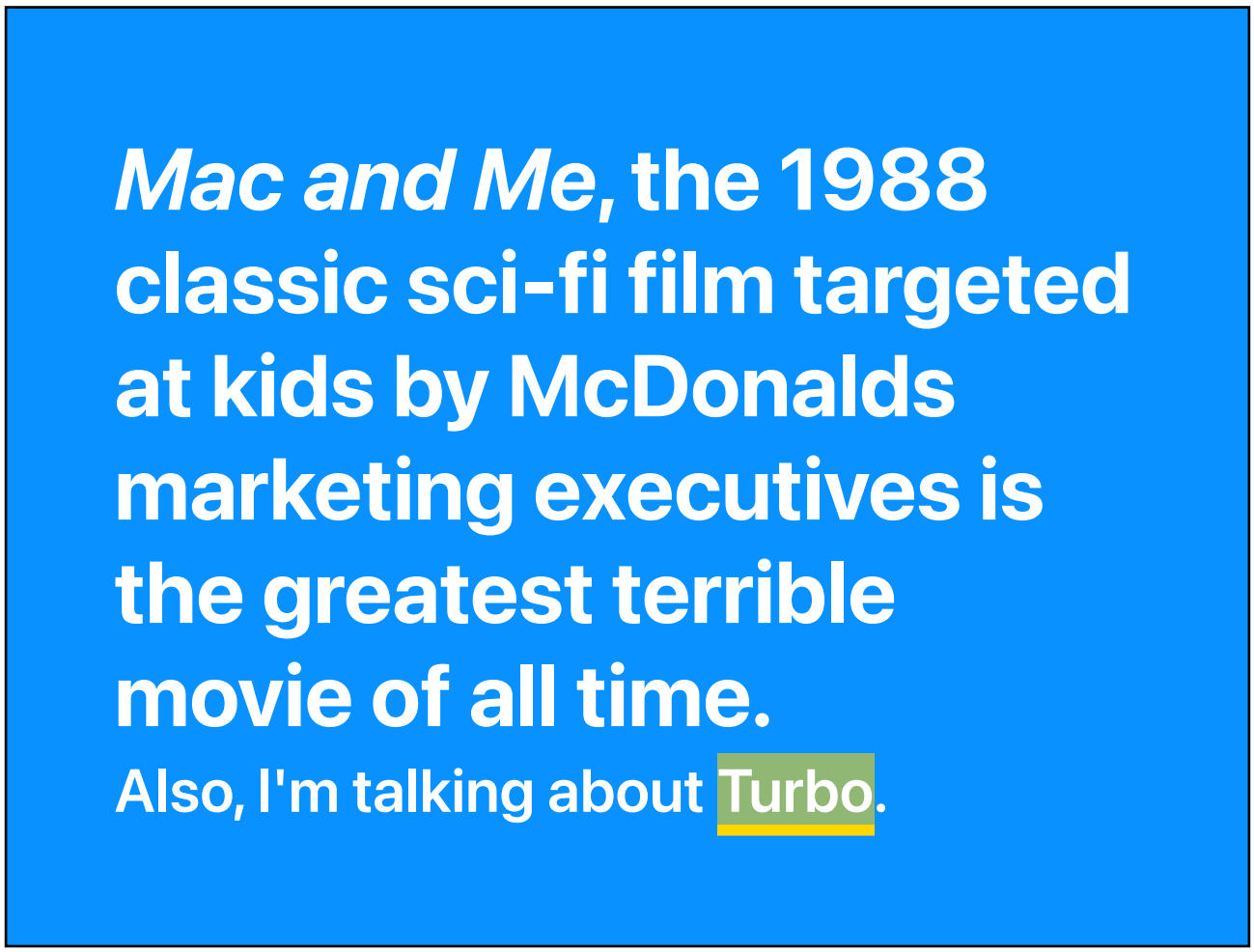
Wednesday July 12, 2023

...Turbo and You!

Thank you for having me, I am real excited to be up here tonight talking with you all.

How about that poster tease?

Do we all know what I am talking about tonight as teased in this poster ...



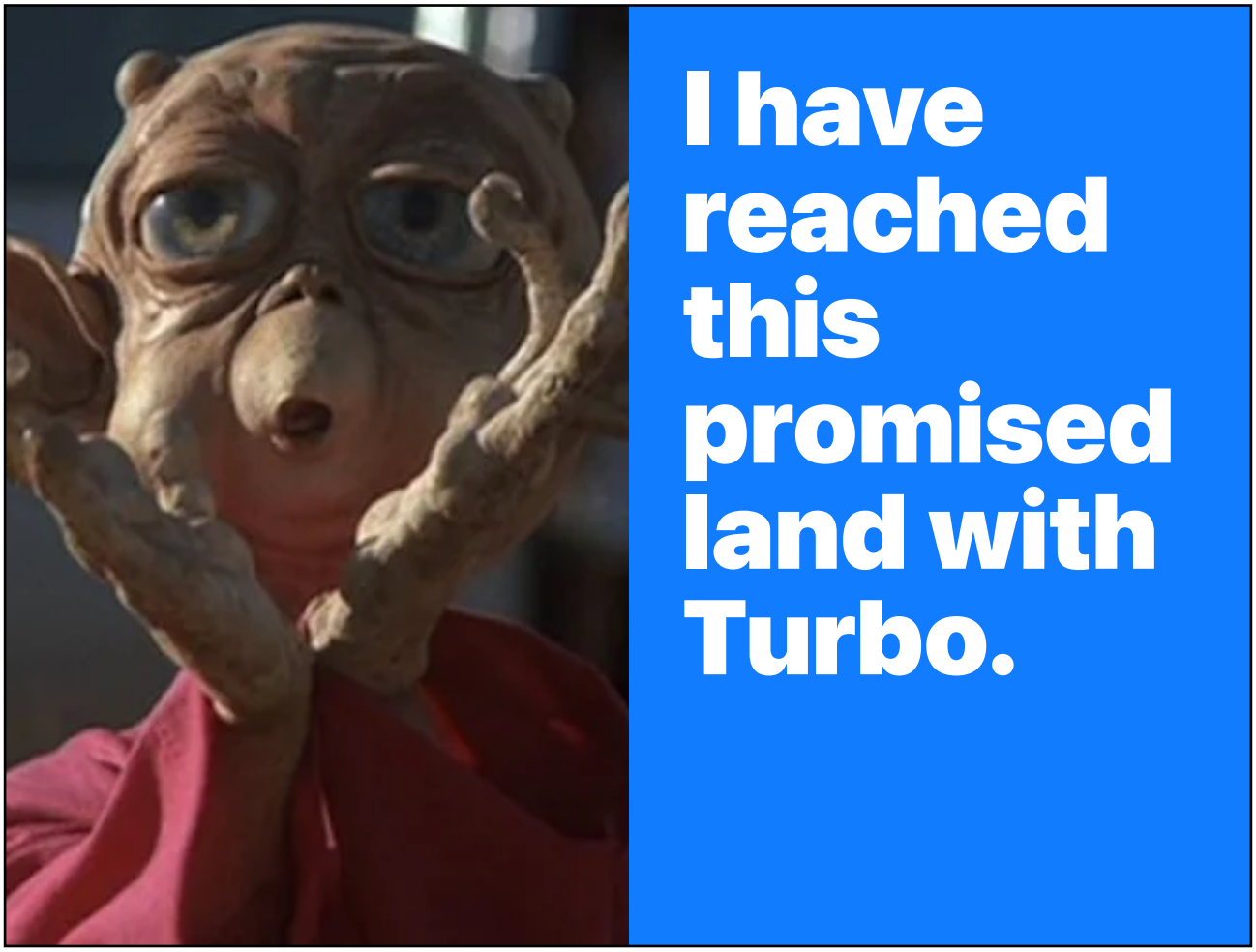
That's right. Mac and Me!

If you can't find tonight educational, hopefully you find it a little entertaining.

Turbo is the Rails promise of building modern web applications that is fast page loading and very little to no Javascript.

More Rails, less Javascript (or at least less complex frontend dependencies is often how I think of things). Sound like a dream eh?

Tonight, I'll be taking apart all the turbo parts from my rails app and talking about what I learned and how it's set me up for success in building native mobile apps with Turbo Native.



But wait a minute...

Wait, who am I and why should you care?

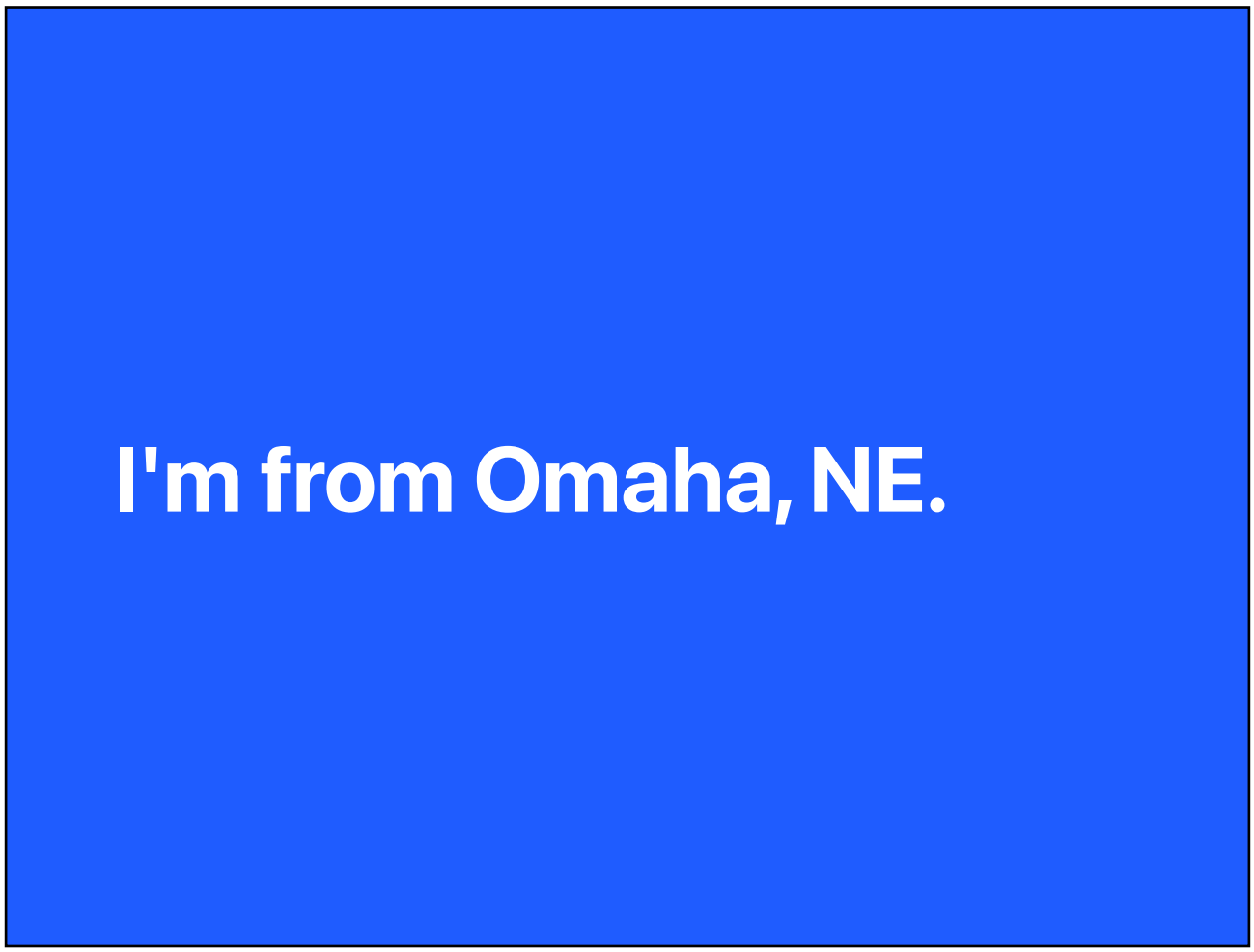
Andy Peters

@andypeters

Hello, I'm Andy Peters!

andypeters.com

- mastodon.social/@andypeters
- twitter.com/andypeters
- github.com/andypeters
- threads.net/andypeters

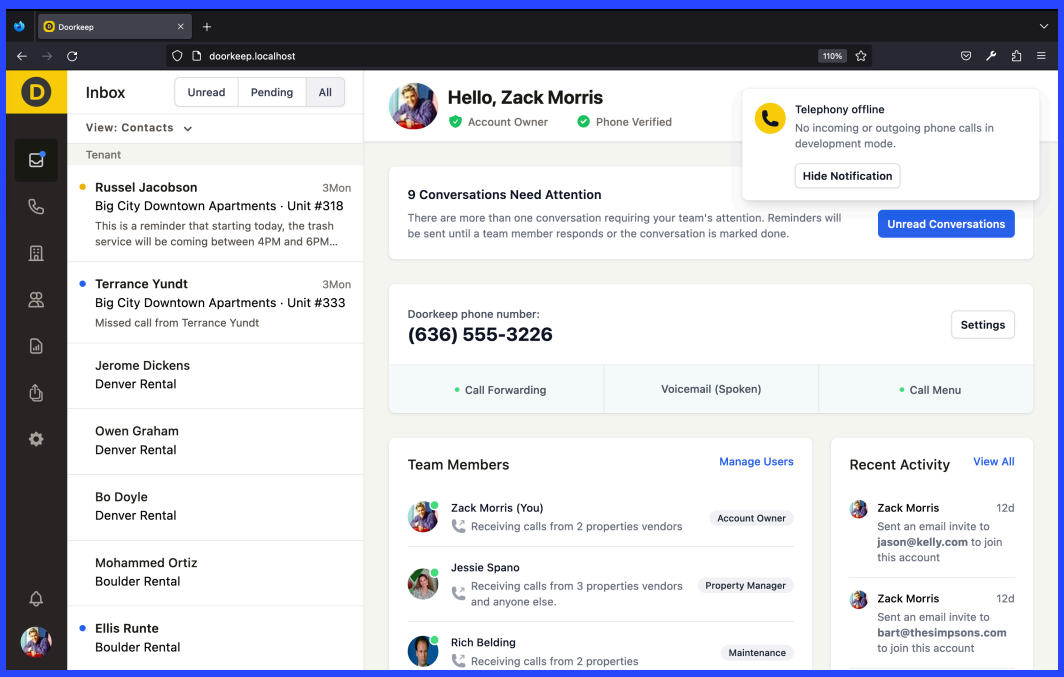


I've been working with Rails since around version 1.2 in various capacities. Good times.

doorkeep.co

Communication software like Google Voice, but designed for anyone managing communications for rental properties.

- Help desk system but designed for property managers.
- Started when I began renting more and got to know the landlords/property-managers. Grew into a small side project then I talked to hundreds of property managers and it continued to grow.
- Bootstrapped product that I have solo-founded. I'm full time on it, but I often take on a part time independent Rails contracts as well.
- I'm here to talk about it as a Rails app and specifically Turbo...



The Doorkeep inbox

Ruby on Rails monolith

Rails 7, Stimulus, Turbo, Sidekiq, TailwindCSS

Postgresql / Redis

Hosted on Render

Doorkeep is a standard full-stack Ruby on Rails monolith app.

What should I talk about at the Boulder Ruby meetup?

Tonight's talk is 100% inspired by a few meetups I attended where everyone was audibly and visibly confused about a lot of topics regarding Turbo and Hotwire. These are problems I hear a lot from you and other Rails developers...

Problem 1

Turbo Documentation could be better.

The main website for all things Turbo (Hotwire) is a good reference but it's not good at getting you started much less taking you beyond getting started.

Problem 2

Not a lot of examples in that there internet world.

It's hard to find across ye' ole internet many excellent Turbo examples.

Problem 3

There's a learning curve.

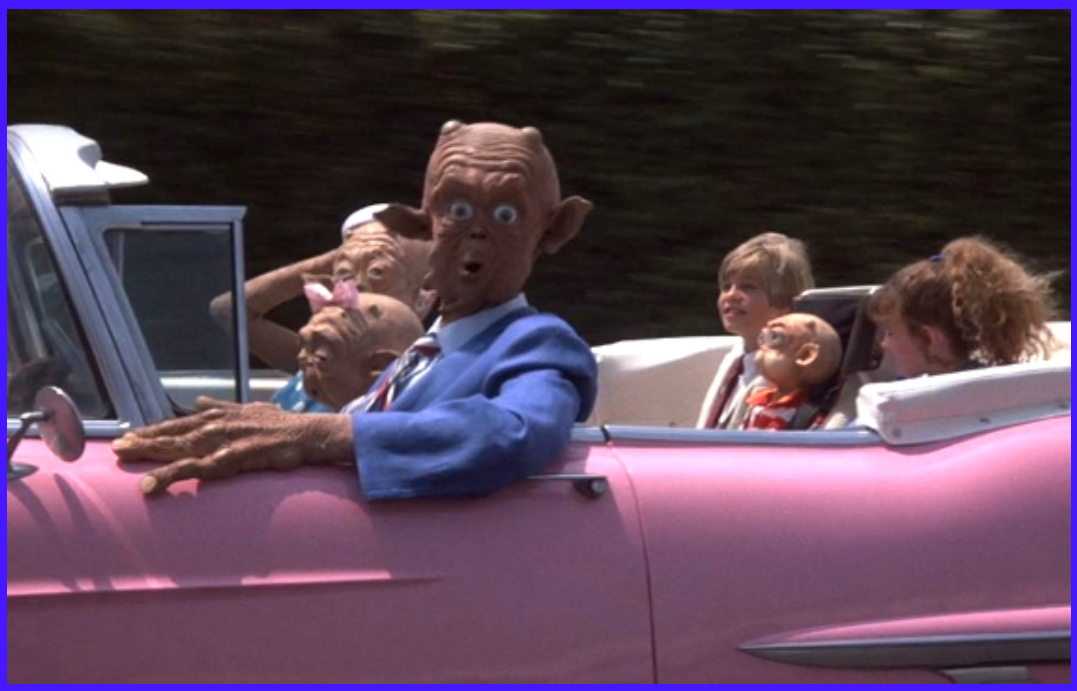
Even though it simplifies a lot of frontend leg work and approaches that all from the Rails ecosystem, Turbo does add an additional level of complexity to your app. In particular if you are learning rails or web development for the first time, it can be challenging.

I learn best by real life examples.

I don't know about all of you, but I learn best by real world examples. This is why I decided to tackle those problems with some show and tell from my app.

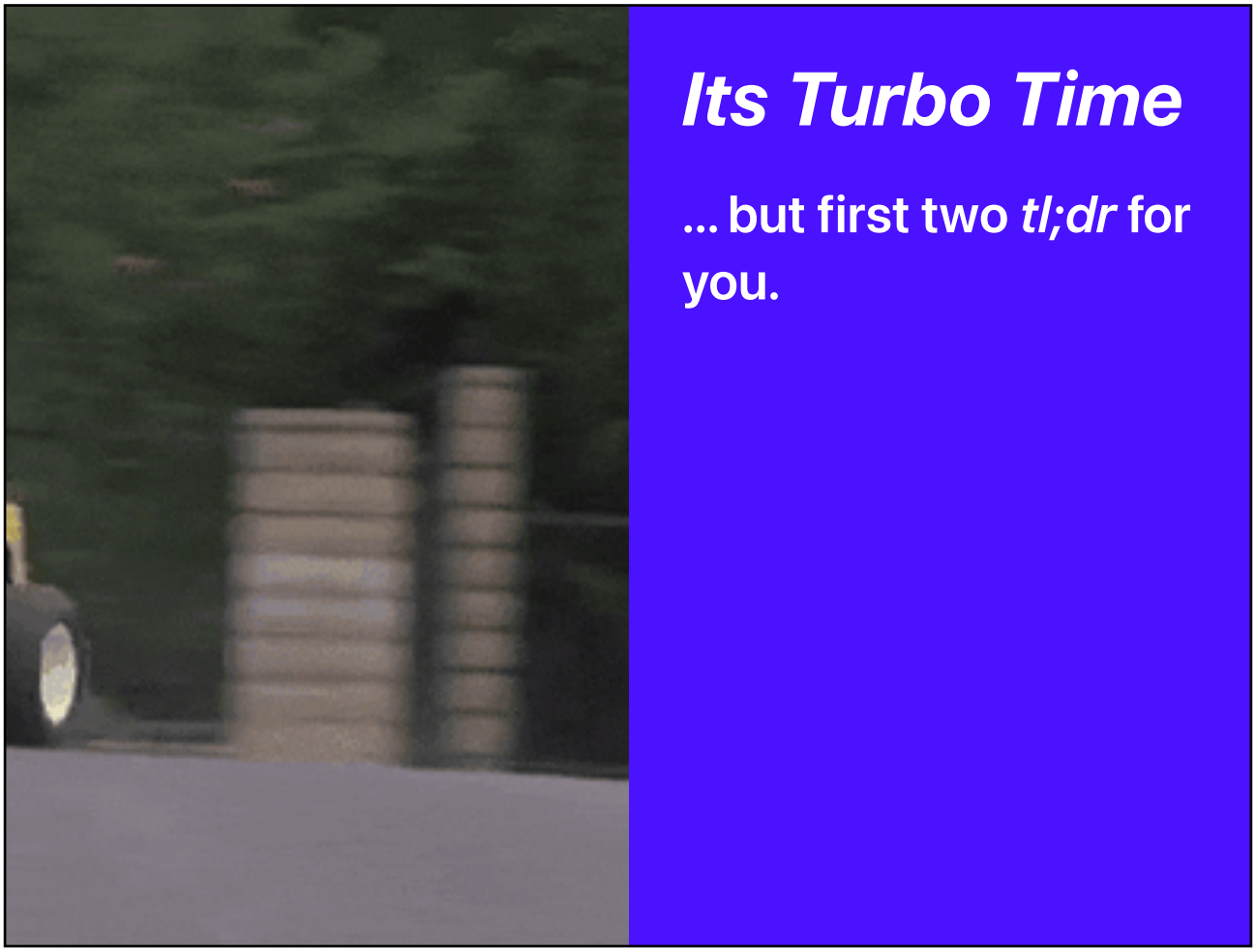
I learned a lot of Turbo by looking at the code for Hey.com back in the day. So that is a little of my approach tonight.

Isn't that gif ridiculous? It's really in the movie. This was Paul Rudd's classic clip for his bit with Conan.



Buckle up. We're have a lot to cover.

So, I'm here tonight to show you bits of Turbo I use a lot, explain how they work, how they work in my app and talk about how it's helped me prep for the native apps with Turbo Native.



I have two quick take-aways for you...

tl;dr:

Are you starting out learning Rails development or have a legacy app with lots going on...

```
format.turbo_stream
```

...will be confusing to debug.

Watch for `TURBO_STREAM` in your logs.

There is already enough to grasp when it comes to working on understanding how to submit a form via HTTP POST, but when adding `format.turbo_steam` to the mix, it'll be harder.

So my suggestion is that if you are learning this Rails thing right now, maybe try to avoid Turbo until you get all the typical forms and navigation stuff figured out.

Does anyone have similar thoughts or experiences?

tl;dr:

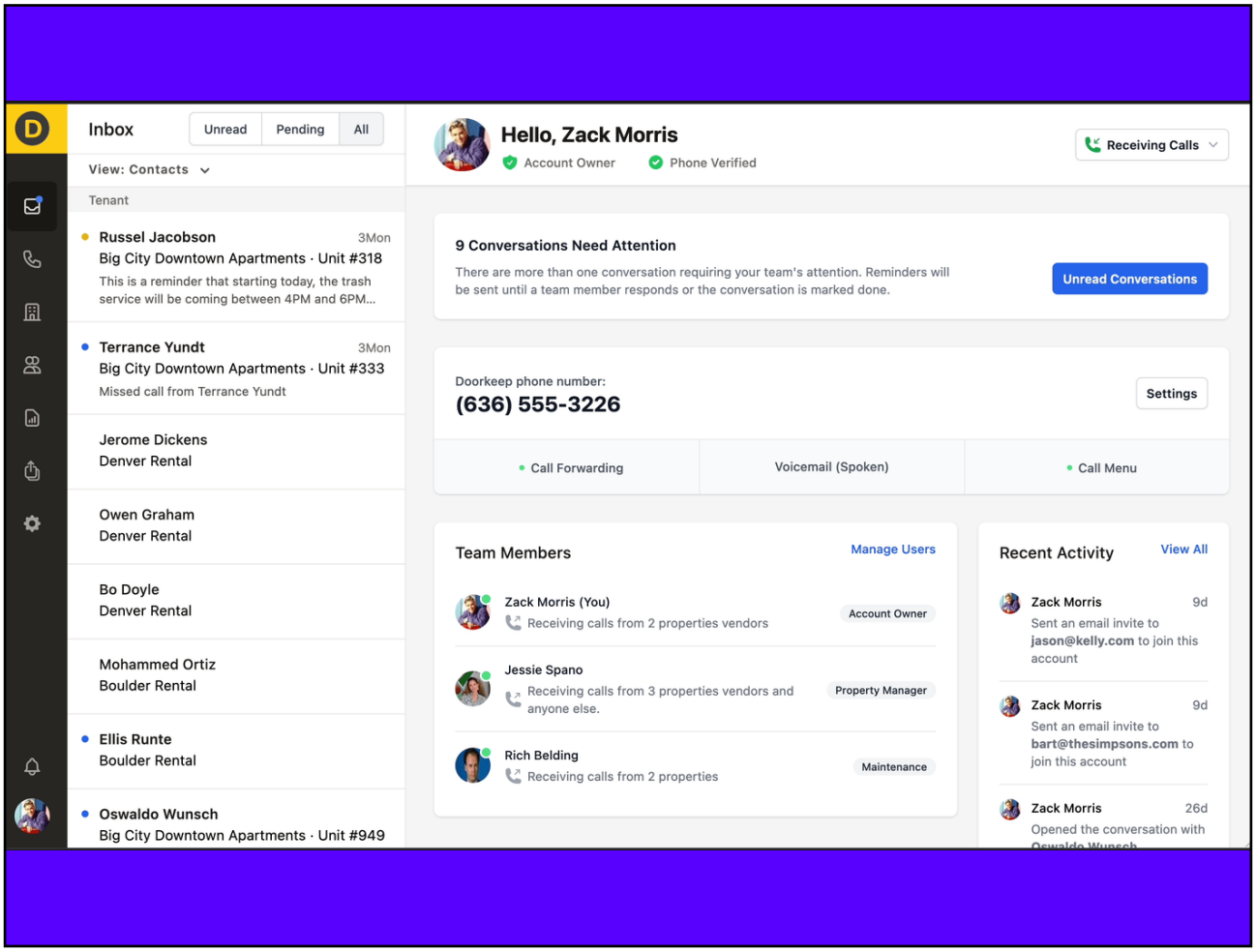
Do have *Turbo Native* dreams?

Make your app design responsive from the start. Or start here first.

For Turbo Native, you will need to ensure your entire front end is responsive. Of course, you don't need a responsive design to take advantage of Turbo Frames or Turbo Streams though.

I use TailwindCSS, it makes it easy but you don't have to use that of course. There are plenty of others or have your awesome design team make one; Bootstrap, Bulma are just to name a few that I know of.

Here's a tiny flex from myself. Can your app does this?



This is Doorkeep in Firefox Developer Edition, from Desktop down to mobile and back.

Alright. Moving on...

What is Turbo?

I should have asked earlier. Raise your hand if you are using Turbo and all Hotwire has to offer semi-regularly or have in the past? Turbo?

😕 (visible confusion)

> Is it Hotwire?
> Is it Hotwired?
> Or is it Turbo?

Who can explain what Hotwire is?

Sarcasm wasn't intended here, the naming seems to be confusing to folks. IDK.

Let's clear it up a bit...

HOTWIRE

*H*TML *O*ver *T*he *W*ire

Hotwire is the set of frontend libraries that enable any of us to build modern web application without using the large frontend frameworks like Vue, React, etc. This lets us Rails developers rely on server rendered HTML (`.erb`, `.haml`, etc) files instead of JSON and Javascript.

Hotwire is the clever acronym for *HTML Over the Wire*.

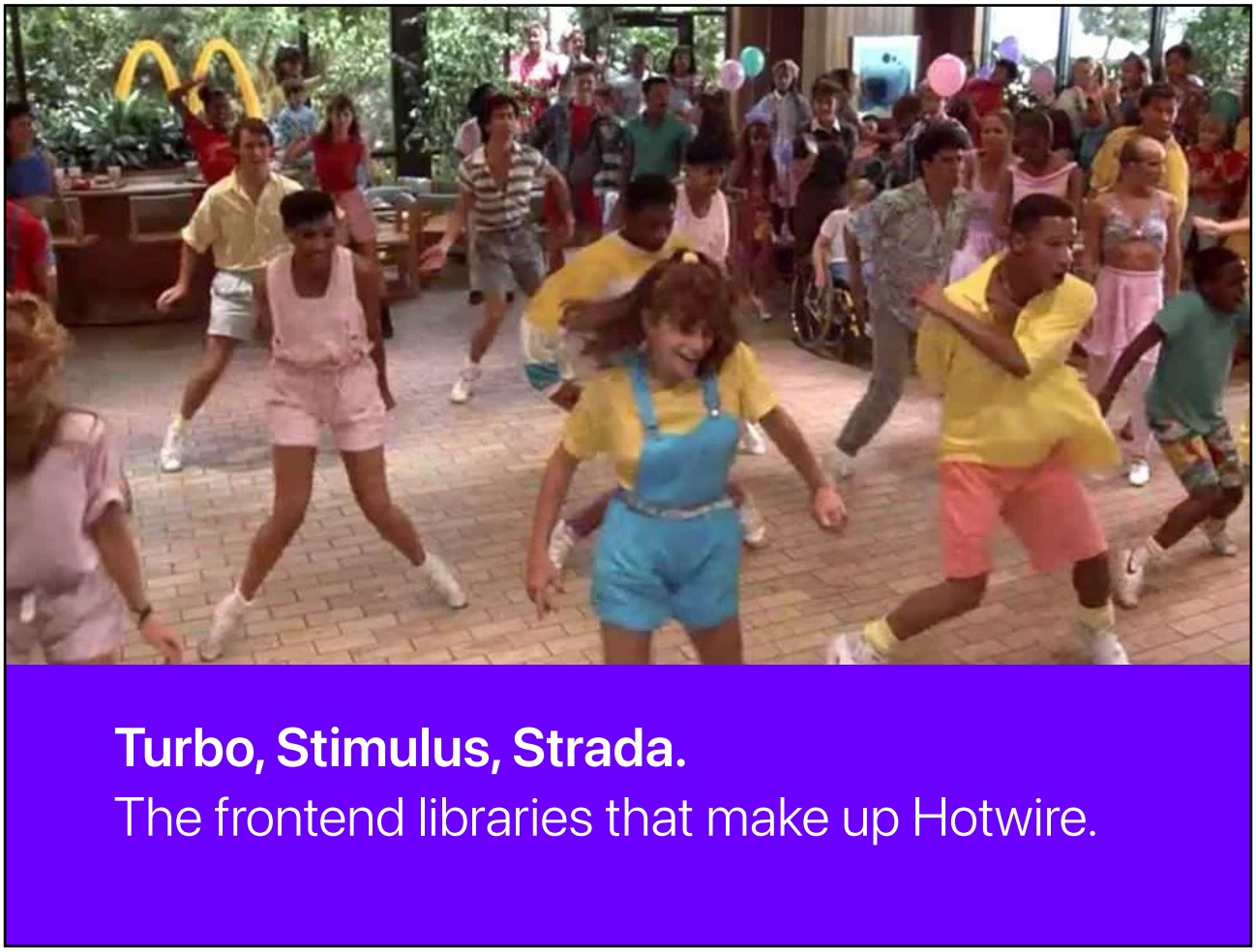
***Hotwired* is just the name of the website.**

https://hotwired.dev

Hotwired is just the name of the website. The entire package of frontend toolsets is called *Hotwire*.

I had always thought it was because `hotwire.dev` was taken, but it does a redirect to `hotwired.dev`. Does anyone happen to know history as to why it was `Hotwired.dev`?

So, what are these frontend libraries that make up Hotwire you ask?



These are the frontend libraries that make up Hotwire are Turbo, Stimulus and Strada.

I think everyone here knows Stimulus by now since it's been around the Ruby on Rails ecosystem for a while. It's now under the Hotwire package of things. That change I believe was welcomed with the version 3.0 release as well.

Stimulus is a lightweight mechanism using which we can attach pieces of JavaScript logic to HTML elements in the form of a controller.

Strada augments Turbo Native by providing a bridge to communicate with native code using JavaScript.

As of today, Strada still hasn't been released. It's been "coming soon" for a while. I don't really know what it really is beyond what I said up there. Hopefully it'll come out, make some things easier and be awesome.

Hope that summary was helpful. As I mentioned before, tonight we are focusing on Turbo.

Turbo Drive

Remember Turbolinks?

From the Hotwire website:

> Turbo Drive is the part of Turbo that enhances page-level navigation. It watches for link clicks and form submissions, performs them in the background, and updates the page without doing a full reload. It's the evolution of a library previously known as Turbolinks.

Turbo Frames

Modern `<iframe>`?

Again, from Hotwire:

> Turbo Frames allow predefined parts of a page to be updated on request. Any links and forms inside a frame are captured, and the frame contents automatically updated after receiving a response. Regardless of whether the server provides a full document, or just a fragment containing an updated version of the requested frame, only that particular frame will be extracted from the response to replace the existing content.

Turbo Streams

`<turbo-stream>`

I don't love the way Hotwire website explains Streams. Here I did my best to write how I think of it:

> Turbo Streams let us to perform a series of CRUD actions on specific DOM elements in our app using a tag. As soon as a tag is added to the document, Turbo will execute it and perform the DOM alteration it defines. These tags can be delivered in a variety of methods such as an HTTP response or WebSockets.

Turbo Native

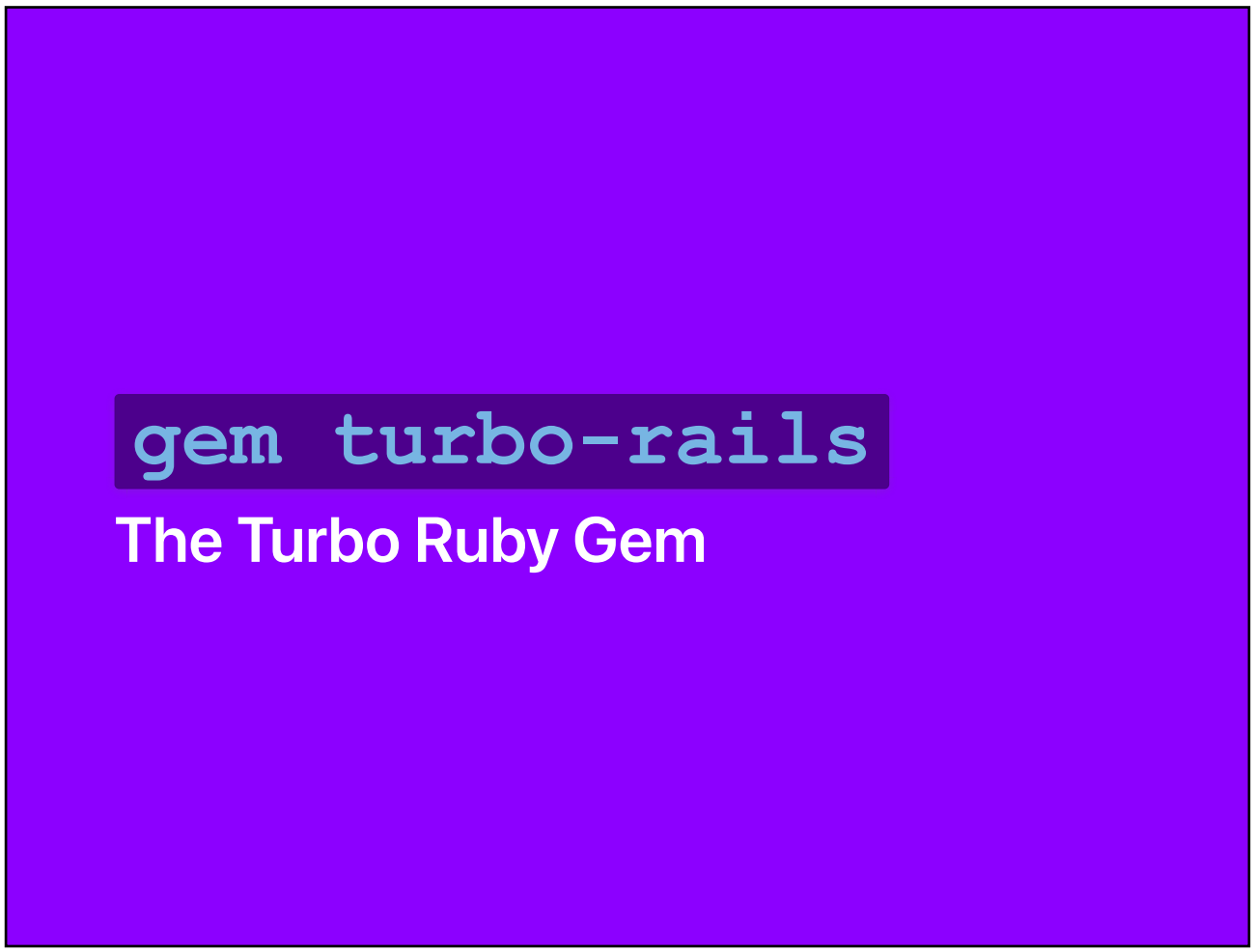
Hybrid, but actually good (enough).

From Hotwire.dev:

> Turbo Native for iOS provides the tooling to wrap your Turbo-enabled web app in a native iOS shell. It manages a single WKWebView instance across multiple view controllers, giving you native navigation UI with all the client-side performance benefits of Turbo.

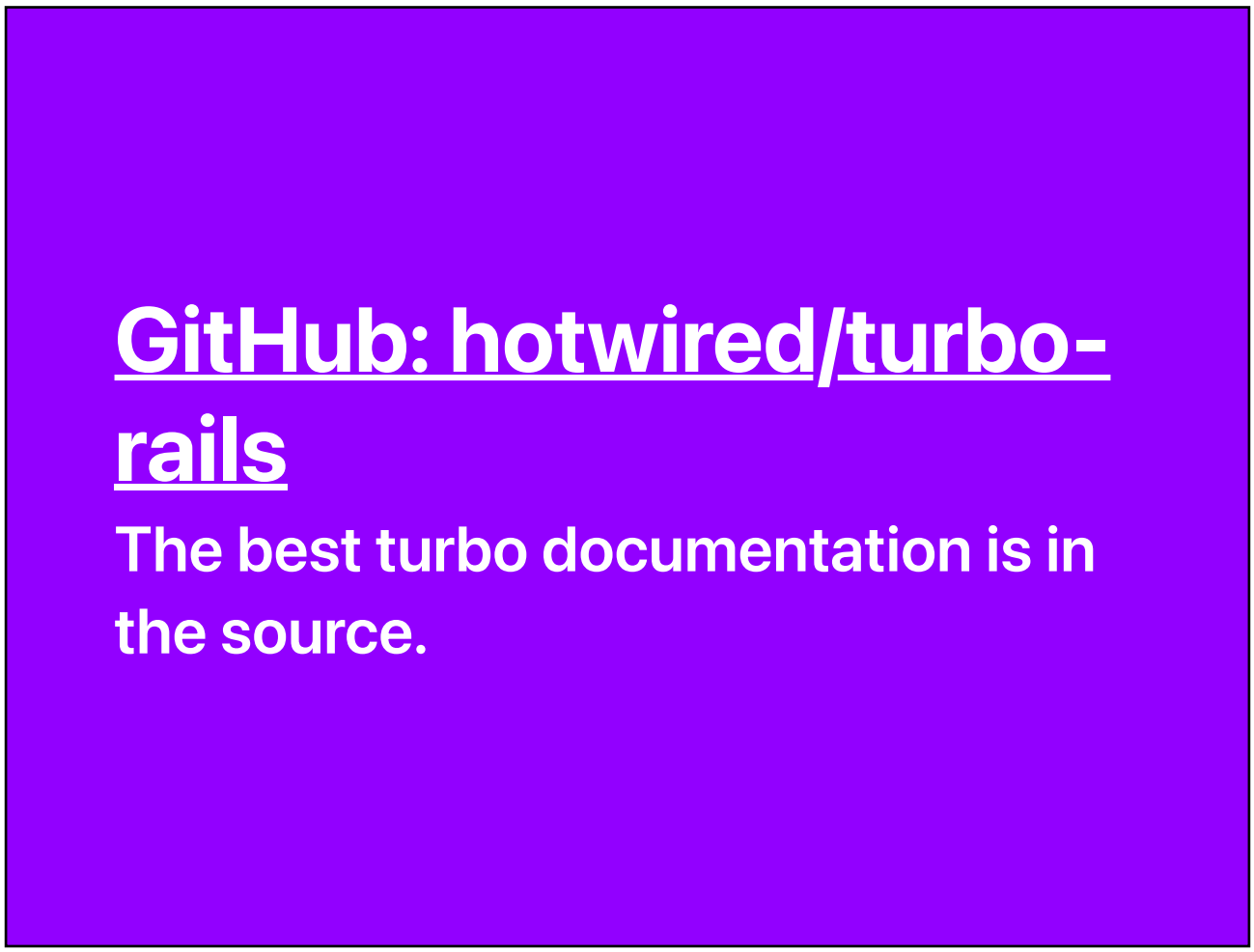
(And when you hit the limits of the web views, you can make a native experience available)

A few Turbo setup notes for you...

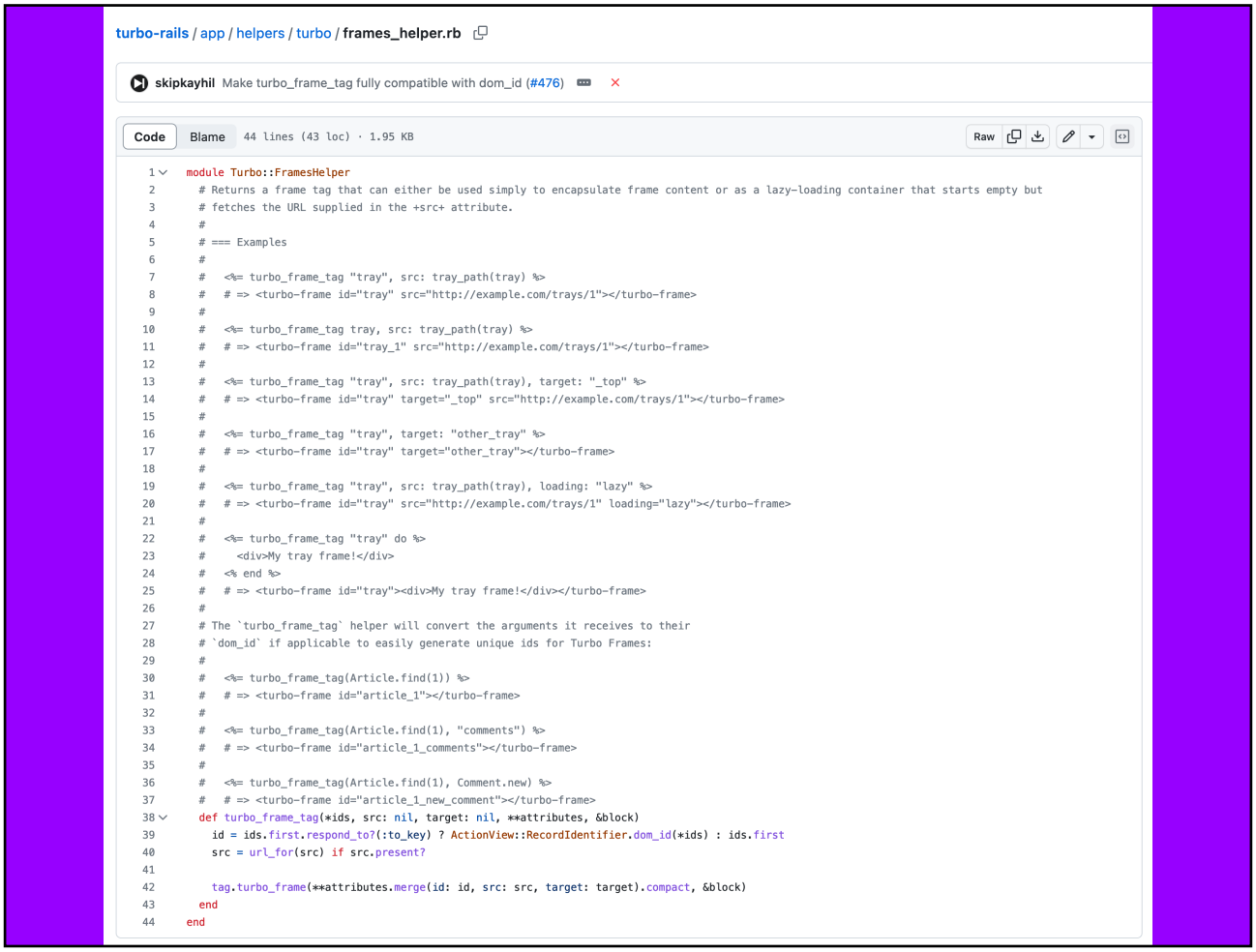


The official Ruby gem for Turbo.

It's automatically configured in Rails 7 apps and of course it can be disabled. Provides rails helper methods to make all Turbo frame or stream actions as Ruby code.



I found the best turbo documentation is in the source.



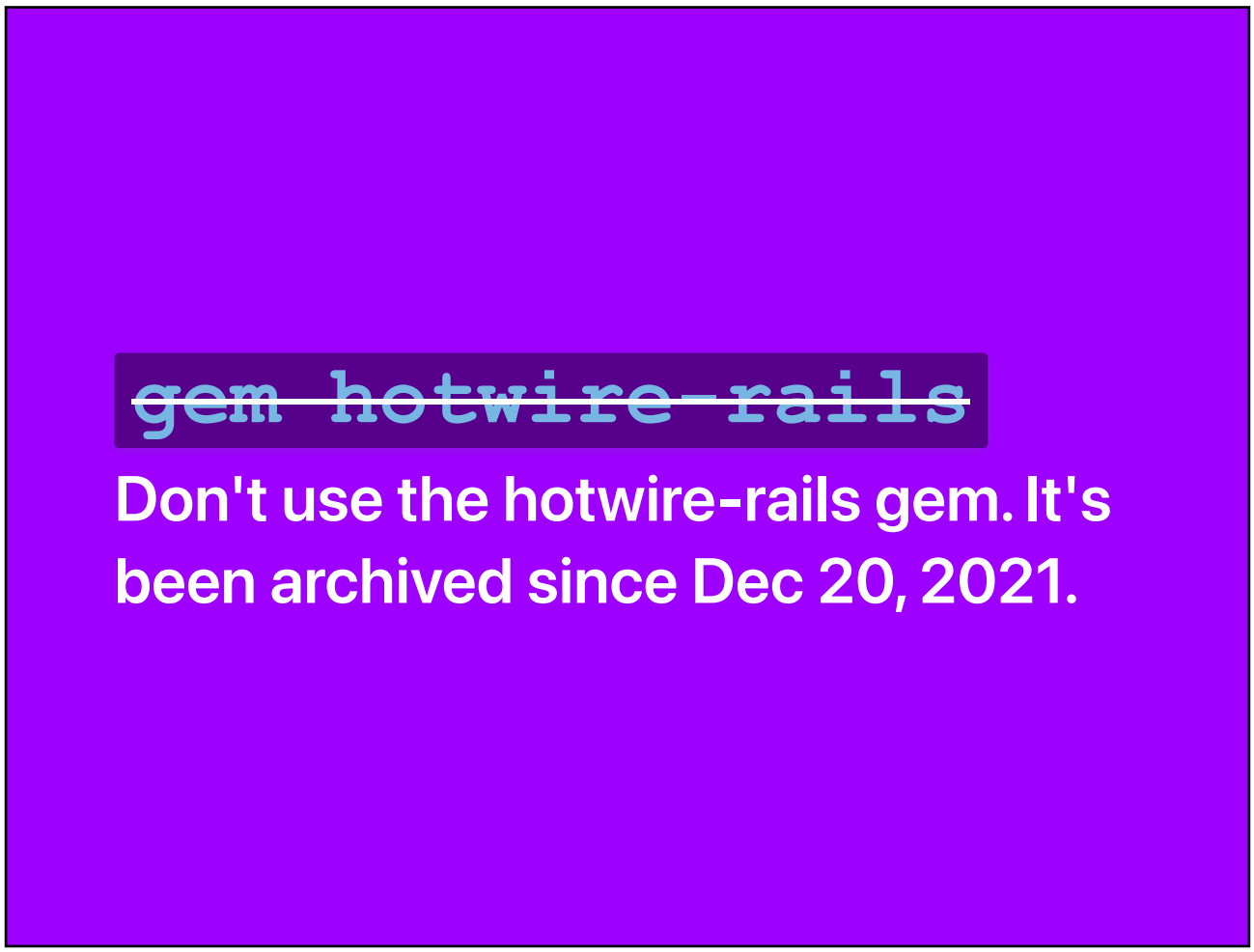
turbo-rails / app / helpers / turbo / **frames_helper.rb**

skipkayhil Make turbo_frame_tag fully compatible with dom_id (#476)

Code | Blame | 44 lines (43 loc) · 1.95 KB | Raw

```ruby
module Turbo::FramesHelper
  # Returns a frame tag that can either be used simply to encapsulate frame content or as a lazy-loading container that starts empty but
  # fetches the URL supplied in the +src+ attribute.
  #
  # === Examples
  #
  #   <%= turbo_frame_tag "tray", src: tray_path(tray) %>
  #   # => <turbo-frame id="tray" src="http://example.com/trays/1"></turbo-frame>
  #
  #   <%= turbo_frame_tag tray, src: tray_path(tray) %>
  #   # => <turbo-frame id="tray_1" src="http://example.com/trays/1"></turbo-frame>
  #
  #   <%= turbo_frame_tag "tray", src: tray_path(tray), target: "_top" %>
  #   # => <turbo-frame id="tray" target="_top" src="http://example.com/trays/1"></turbo-frame>
  #
  #   <%= turbo_frame_tag "tray", target: "other_tray" %>
  #   # => <turbo-frame id="tray" target="other_tray"></turbo-frame>
  #
  #   <%= turbo_frame_tag "tray", src: tray_path(tray), loading: "lazy" %>
  #   # => <turbo-frame id="tray" src="http://example.com/trays/1" loading="lazy"></turbo-frame>
  #
  #   <%= turbo_frame_tag "tray" do %>
  #     <div>My tray frame!</div>
  #   <% end %>
  #   # => <turbo-frame id="tray"><div>My tray frame!</div></turbo-frame>
  #
  # The `turbo_frame_tag` helper will convert the arguments it receives to their
  # `dom_id` if applicable to easily generate unique ids for Turbo Frames:
  #
  #   <%= turbo_frame_tag(Article.find(1)) %>
  #   # => <turbo-frame id="article_1"></turbo-frame>
  #
  #   <%= turbo_frame_tag(Article.find(1), "comments") %>
  #   # => <turbo-frame id="article_1_comments"></turbo-frame>
  #
  #   <%= turbo_frame_tag(Article.find(1), Comment.new) %>
  #   # => <turbo-frame id="article_1_new_comment"></turbo-frame>
  def turbo_frame_tag(*ids, src: nil, target: nil, **attributes, &block)
    id = ids.first.respond_to?(:to_key) ? ActionView::RecordIdentifier.dom_id(*ids) : ids.first
    src = url_for(src) if src.present?

    tag.turbo_frame(**attributes.merge(id: id, src: src, target: target).compact, &block)
  end
end
```

For real, I used this all the time. Props to the team from me, thank you. This is a screenshot from today from one of the helpers.



This is likely old or "non" news to everyone. However, if you stumble on a tutorial or older code examples that includes this gem you'll want to replace it with `turbo-rails`.

Some people talking Turbo a lot to follow

Names	Github
Nate Hopkins	@hopsoft
Macro Roth	@marcoroth
Julian Rubisch	@julian_rubisch
Ayush Newatia	@ayushn21
Chris Oliver	@excid3
Joe Masilotti	@joemasilotti
Andrea Fomera	@afomera

Just some shout outs to people I do not know and never have met but I have learned by following code or on twitter or mastodon or whatever. There are others I'm sure and I am sorry I forgot you!

Websites

- Hotwired
- GitHub hotwired/turbo-rails
- GoRails
- GoRails Discord #hotwire
- Hotrails
- The Rails and Hotwire Codex
- The Hotwire Club

`Hotwired.dev` is a good reference and so is the GitHub repos. But these also do an excellent job diving deep into real world development with Turbo. I've learned a lot from all of these.

I wanted to plug Hotwire Club prematurely because it sounds packed full of good information even though I have not joined it yet.

Gems / Etc I like

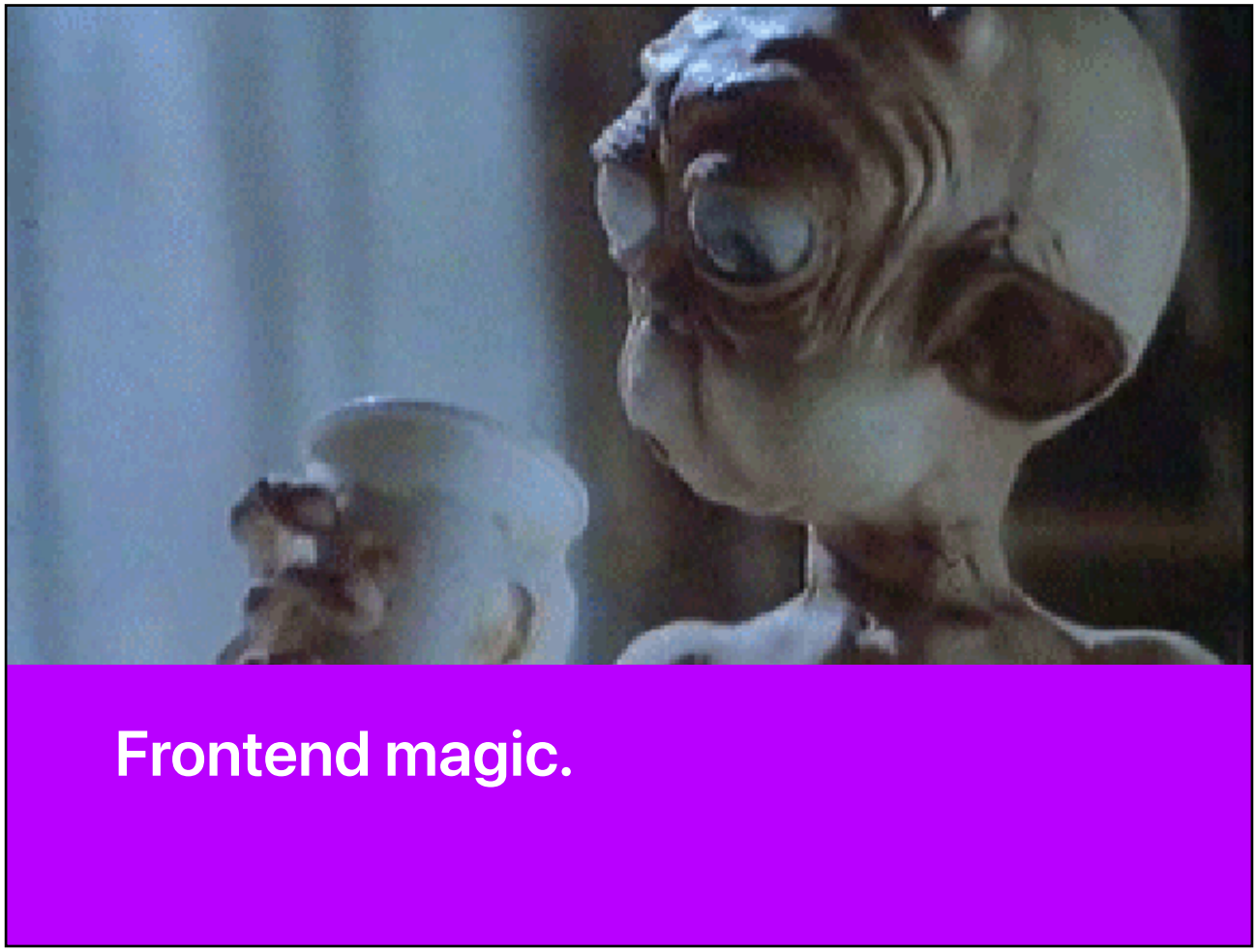
Gem	Description
hotwire-livereload	Automatically reload Hotwire / Turbo when app files are modified.
🤷	I thought I had more.

Livereload is excellent because if you make a change to your stimulus controllers or other javascript or the sorts it'll be sure to reload your page for you. Probably does more? Its a simple helpful development thing.

I guess I thought there was more.

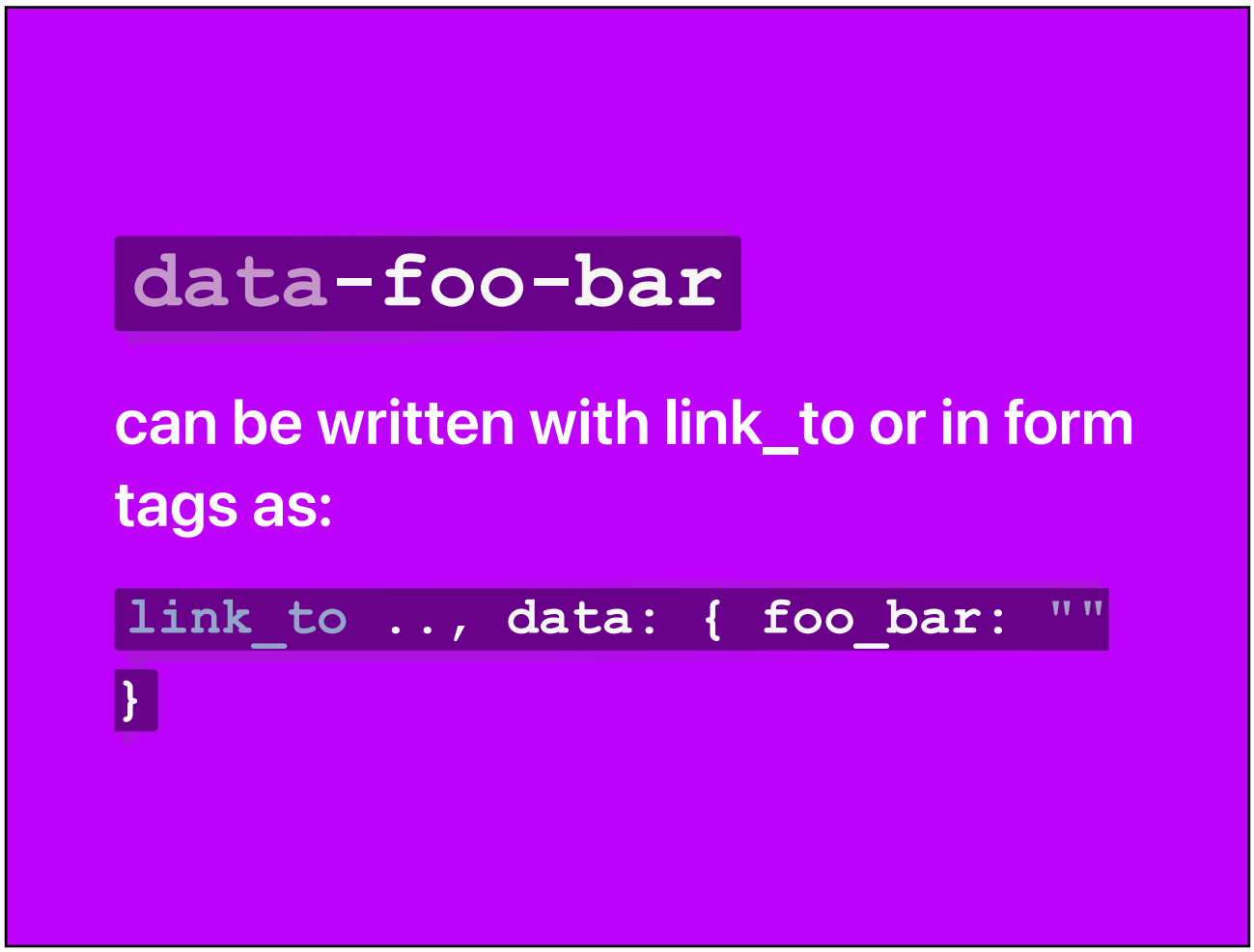
Turbo Drive

Remember the summary on Turbo Drive from a few minutes ago? Good! Let's dive into Turbo Drive quick...



Honestly, I don't knowingly use anything labeled "Turbo Drive" in my app, except just a few items. It's a magical frontend library that does a lot of stuff and powers what Turbo Frames and Streams can do; which has most of my code efforts in.

I just wanted to clarify that I am not directly manipulating anything in javascript for Turbo Drive (that I know of right now).



Just wanted to call out a few development tips I learned I guess.

- Turbo drive has a lot of features and options listed in the documentation of the site. Remember when you are in your Rails app, you are using the `turbo-rails` gem in your templates.
- I wanted to point out that when you see options listed in Turbo Drive documentation the "data" tag can be interpreted as I show right here. I used the link_to method as an example.

*I was surprised how long it took me to realize this. *

For quick confirmations.

`data-turbo-confirm`

```
link_to ..., data: {
turbo_confirm: "" }
```

As an example from the previous slide

> This tag is helpful for a quick Javascript confirmation dialog. This is how you do it. You'll see that turbo drive lists it as `data-turbo-confirm`.

Turbo Frames

Here we go...

Thought process on Turbo Frames.

Started with smaller forms or lists first.

Generally speaking, how I approached decomposing Doorkeep from a large single page load (with lots of Ajax related code) was by working with simple forms inside the app or lists first. Then working my outward to other parts of the app. For example: I wanted to make adding a comment load fast or I wanted to make changing the order of a conversation change fast.

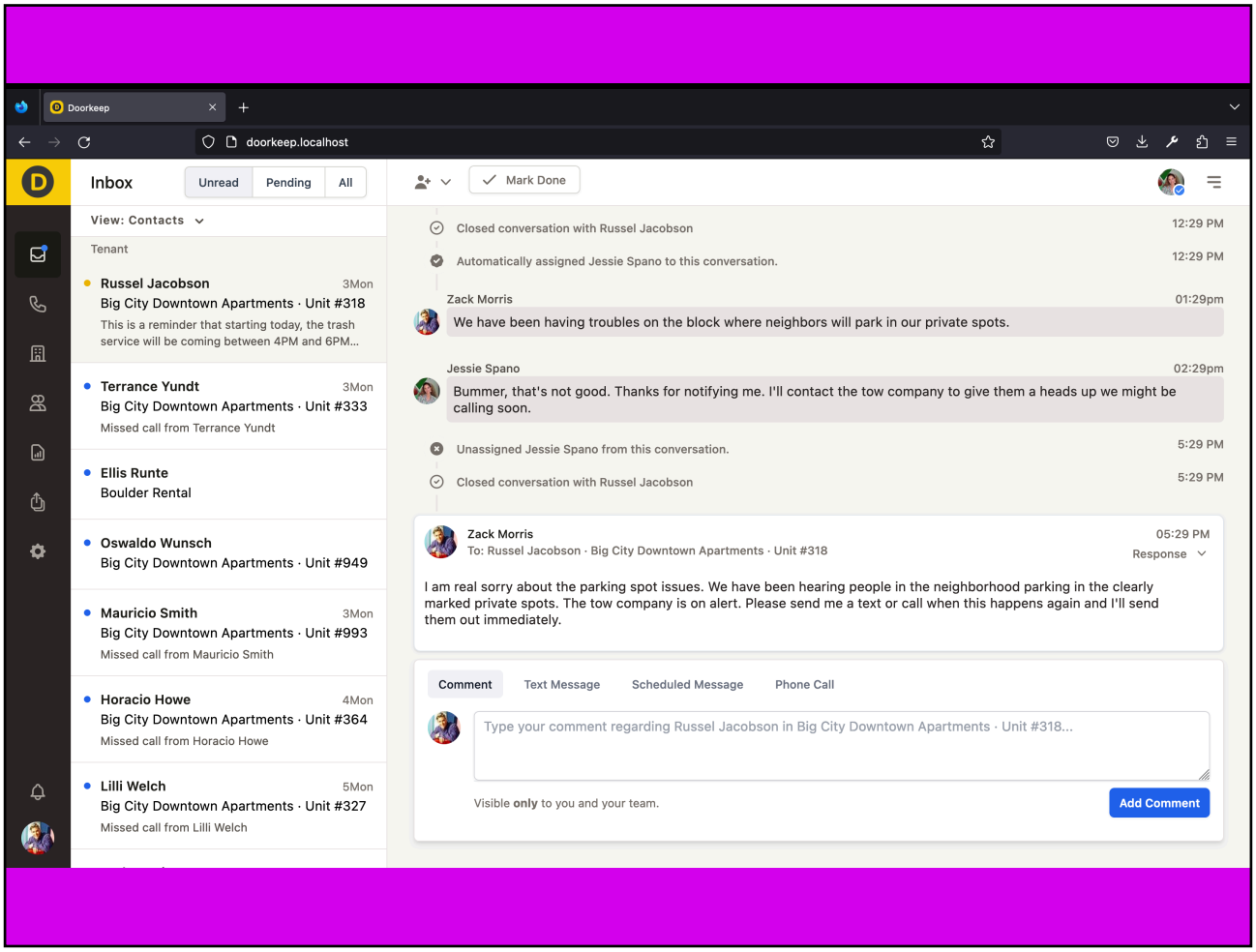
Let's talk about the frame tag code...

turbo_frame_tag

```
turbo_frame_tag "foo"
turbo_frame_tag :foo
turbo_frame_tag @foo
<%= turbo_frame_tag @foo do %>
  <%= render "form" %>
< end %>
turbo_frame_tag tray, src:
tray_path(tray)
turbo_frame_tag tray, src:
tray_path(tray), loading: "lazy"
```

Remember, As long as we are using the `turbo_frame_tag` when we first render the page. And the response from the server uses the exact same turbo frame tag and id, then rails will just replace the content for us. No javascript required.

I use many frames inside this app. Let me break them down for you...



Here is the main Doorkeep inbox layout.

I have some silly sample data loosely based on Saved by the Bell with a bunch of randomly generated people (thanks to the Faker gem) and random conversations.

Let's talk about how the app is currently divided up into frames...

4 core frames.

- "application"
- "secondary"
- "main"
- "notifications"

Okay, they're four core frames that divide up the application. (Remember this is fluid and may change over time)

"Application" frame encompasses the entire app. This wasn't needed until the "notifications" frame.

The "notifications" frame are pop-up style notifications. For example: A notification to answer a phone call or that another user closed a conversation that you need to know about.

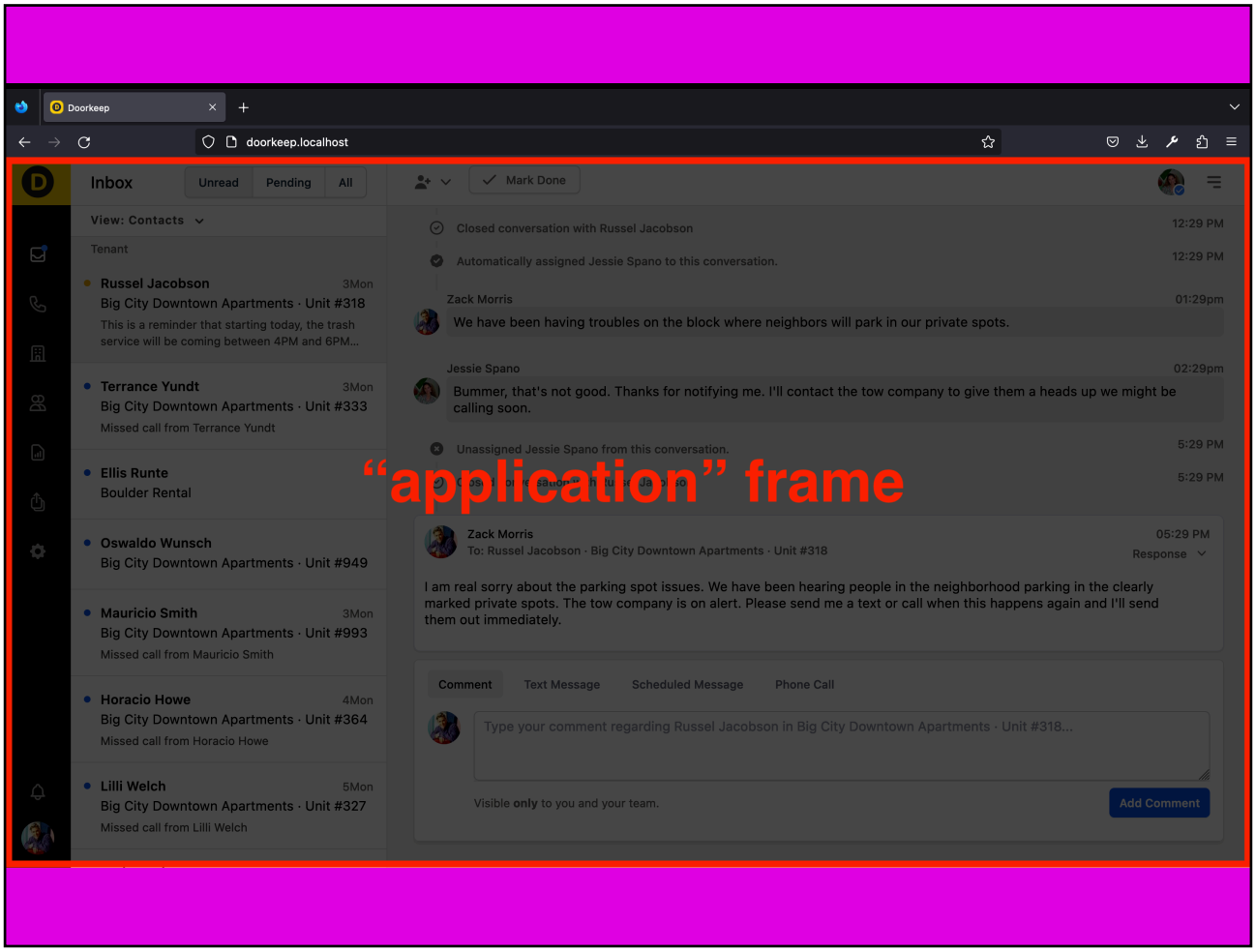
The "main" frame is everything off to the right. This is either the entire conversation frame, contact details, property details, etc. The user spends most of their interactions here.

The "secondary" frame is for secondary navigation. This will change based on the primary navigation. For example: "View conversations inbox" or "Contacts" or "Properties" or "Settings".

Conversation sub frames

- "header"
- "messages"
- a single "message"
- "compose" (new message, comment, etc)

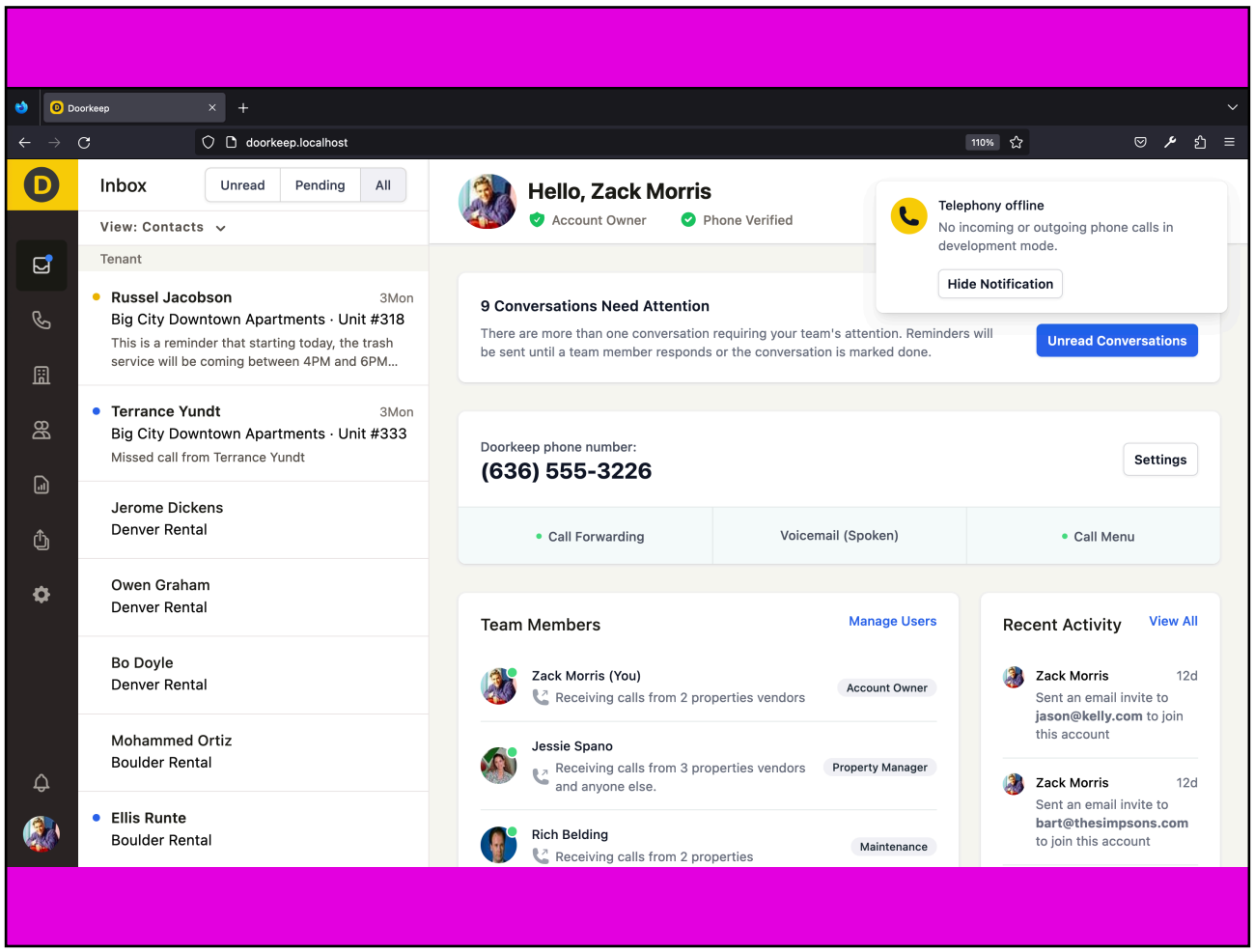
Sub-frames for the Inbox view will be under "Main".



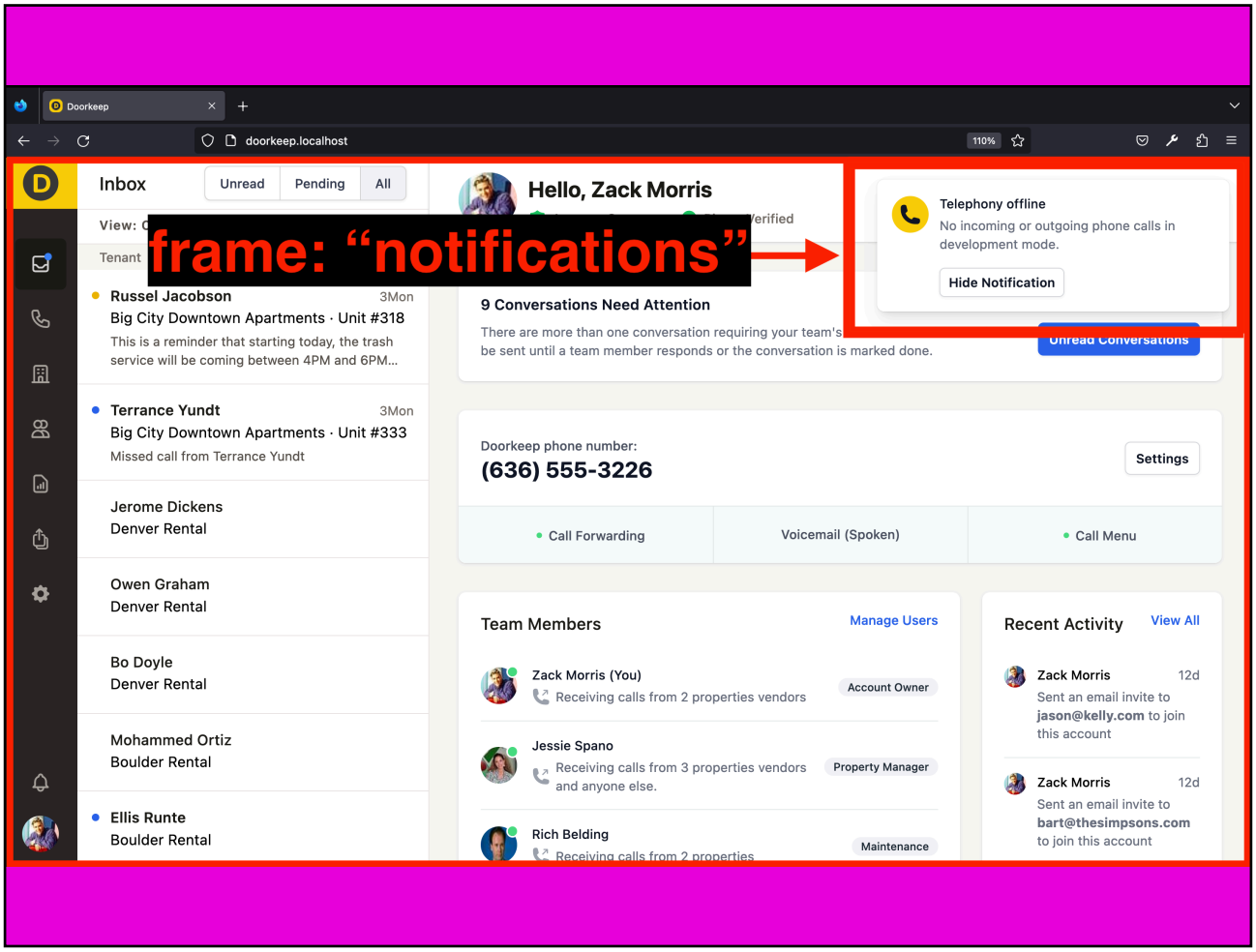
Let's start by talking about the Application frame.

I have a large application frame because there are two parallel sub-frames: "main" and "notifications".

I mentioned before, but "main" frame manages all the normal user interface of navigation, forms, etc.
"Notifications" as the name indicates are pop-up style notifications on the right of the screen.



Here they are. This is the application in Offline mode. The inbox is what you've seen but there is a notification in the top right.



As you can see, there is a notification alerting us that we are in offline / development mode and there is no phone calls. For what it's worth, this was triggered via Stimulus. Of course, there are other notifications that could perhaps be displayed here. They are simply a hidden `<div>` inside the this frame. Turbo Streams will broadcast a message to this notification stream and it'll go from hidden to displayed. Because some of them are interactive, I wanted it to be in its own Frame.

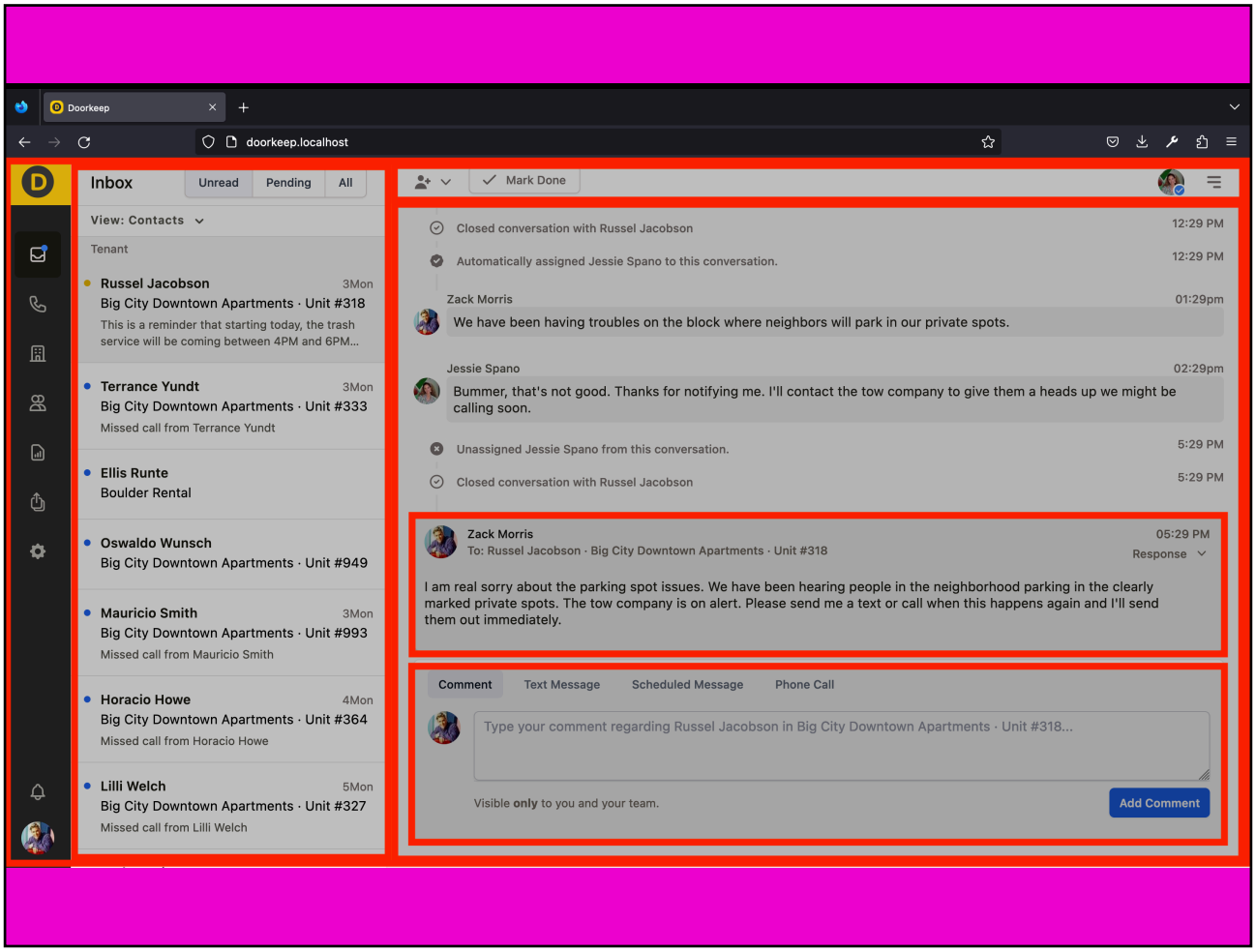
I think I could do an entire talk on how I use the "notifications" frame in conjunction with Turbo Streams. I won't be going into this too much today. Maybe another time?

```
<%= turbo_frame_tag "application",
data: { turbo_action: "advance" } do
%>
<header></header>
<main>
  <%= turbo_frame_tag "main" do %><%
end %>
  <%= turbo_frame_tag "notifications"
do %><% end %>
</main>
<% end %>
```

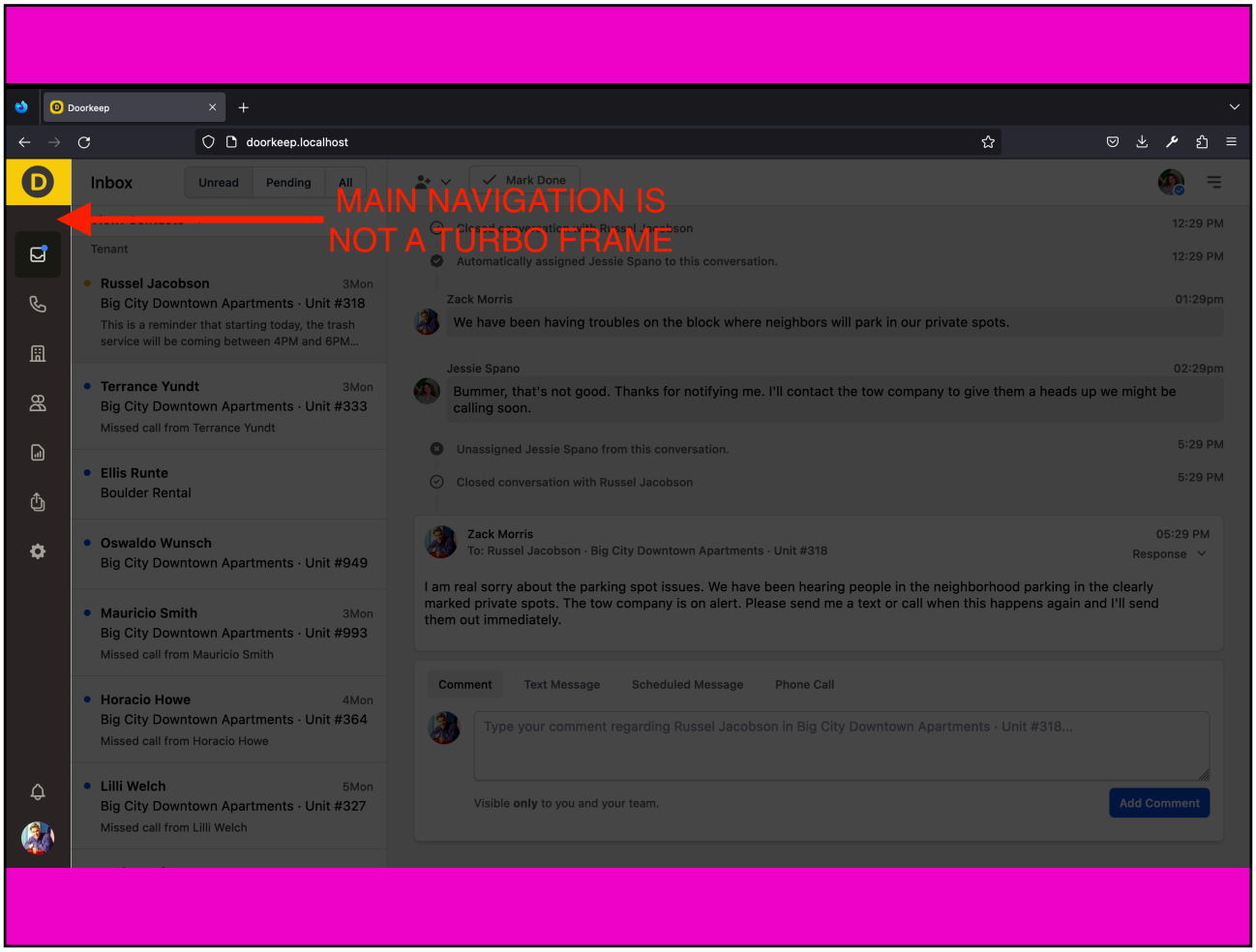
layouts/application.html.erb

Here's some overview code that hopefully illustrates how those previous screenshots work. Notice the `turbo-action-advance`. Let's talk about that.

When that is enabled, the links clicked within that frame update the history of the browser. Without it, the browser history will stay the same as it was when you first loaded. That could be root or the url you click on in an email. So I have the primary navigation advance in the frame.

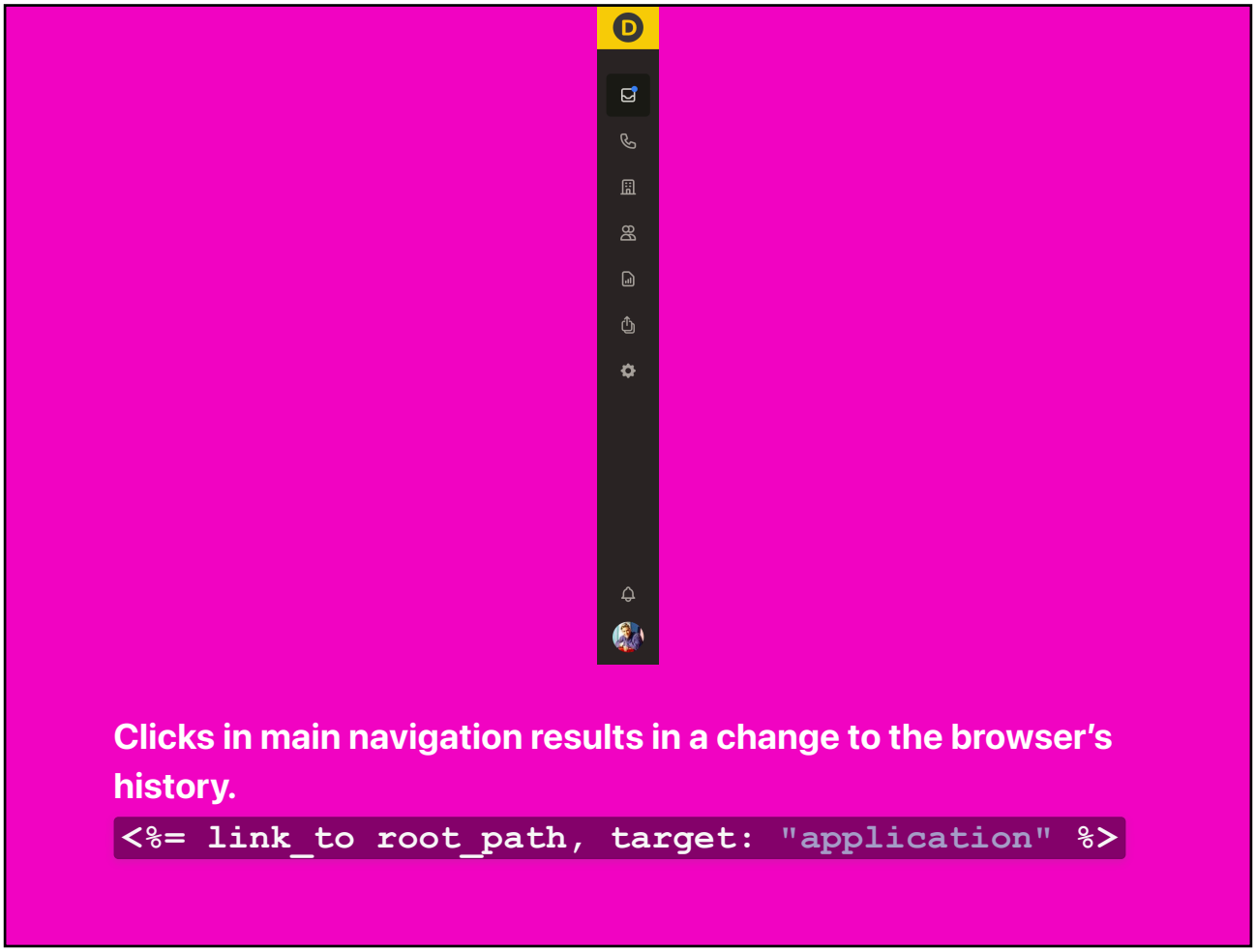


Look at all of those boxes! Hope that illustrates the frames for you in the inbox view.

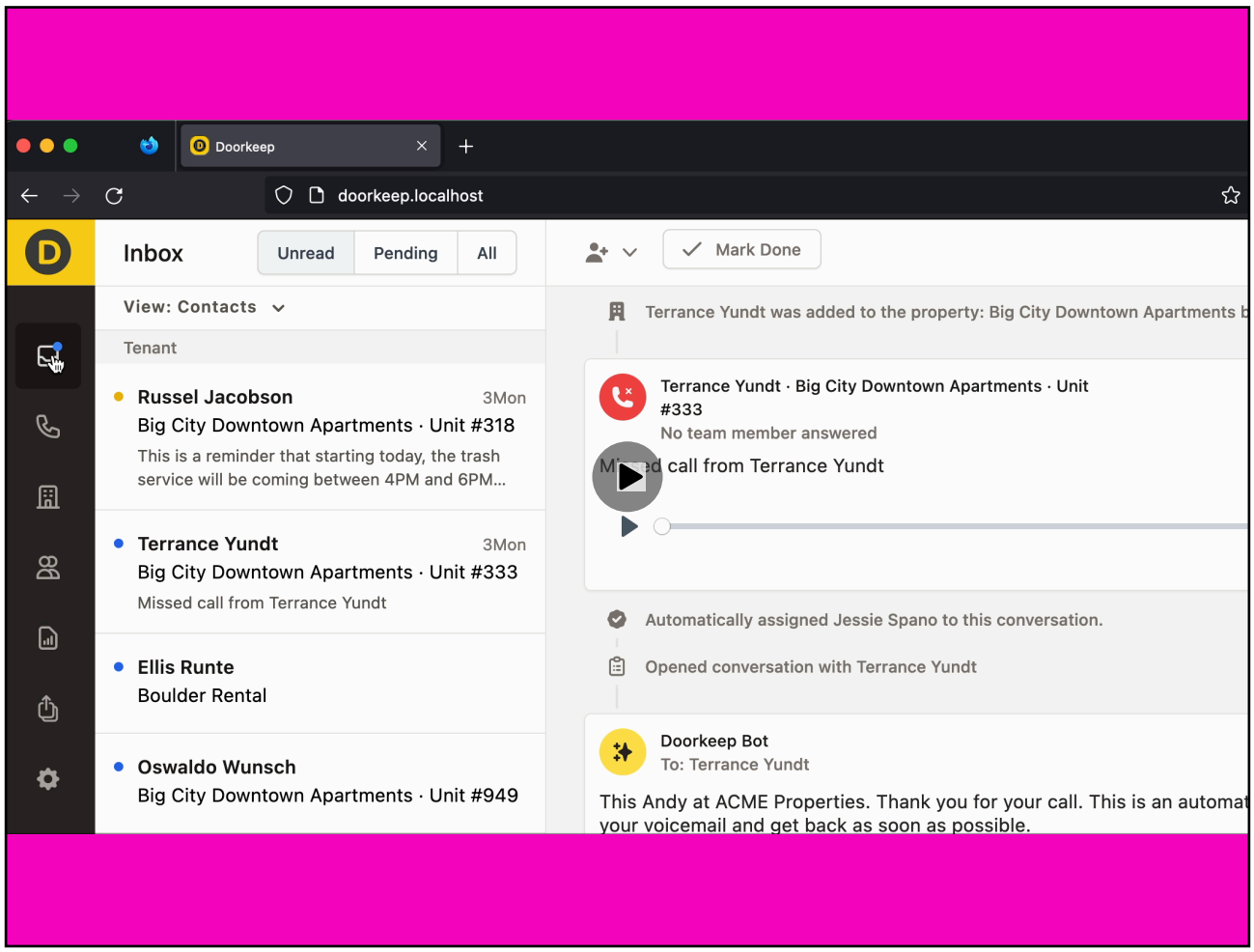


The main navigation bar is not a frame (at the moment).

Why? I only want to change the result of the overall "application" frame from this navigation. I kind of think of it as its the top most navigation for "application" frame so it controls it.

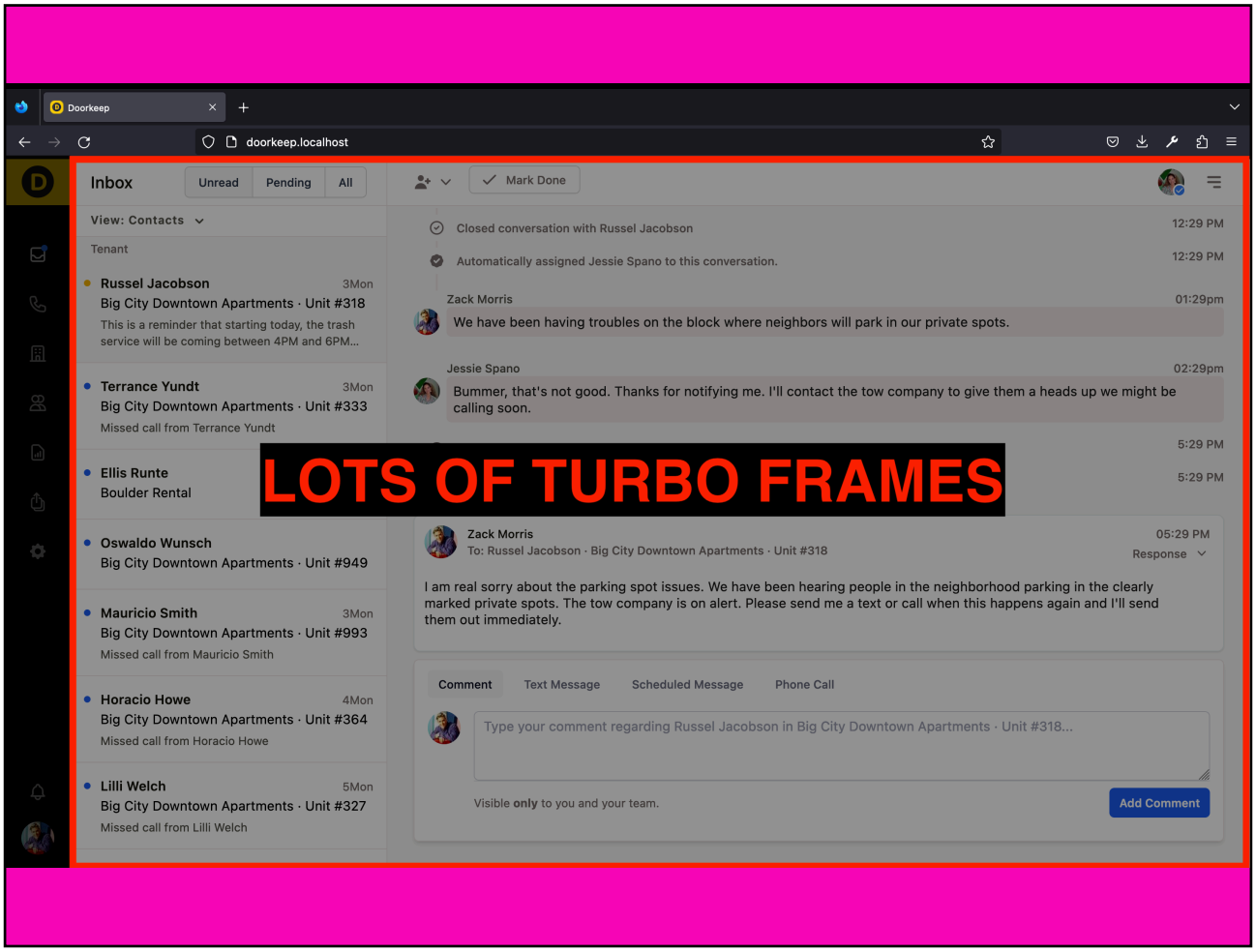


By default, navigation within a frame will target just that frame. Using `target:` here I can drive another named frame by setting the target to that id. In this case it's ensuring everything happens in the application frame from the top most navigation.



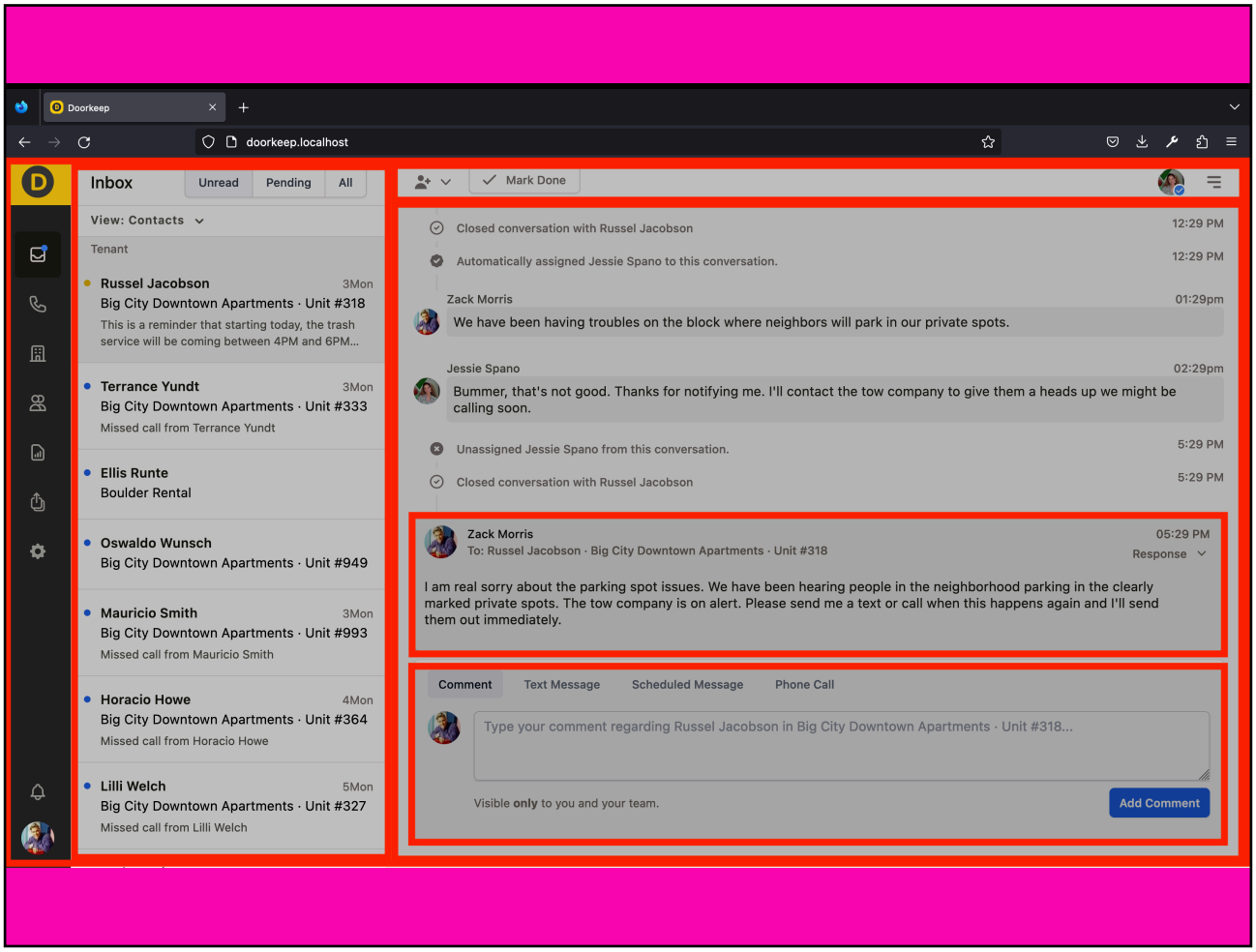
Here's a little video showing clicking around updating the application frame and the browser history (because of the advance option).

Okay, moving on...

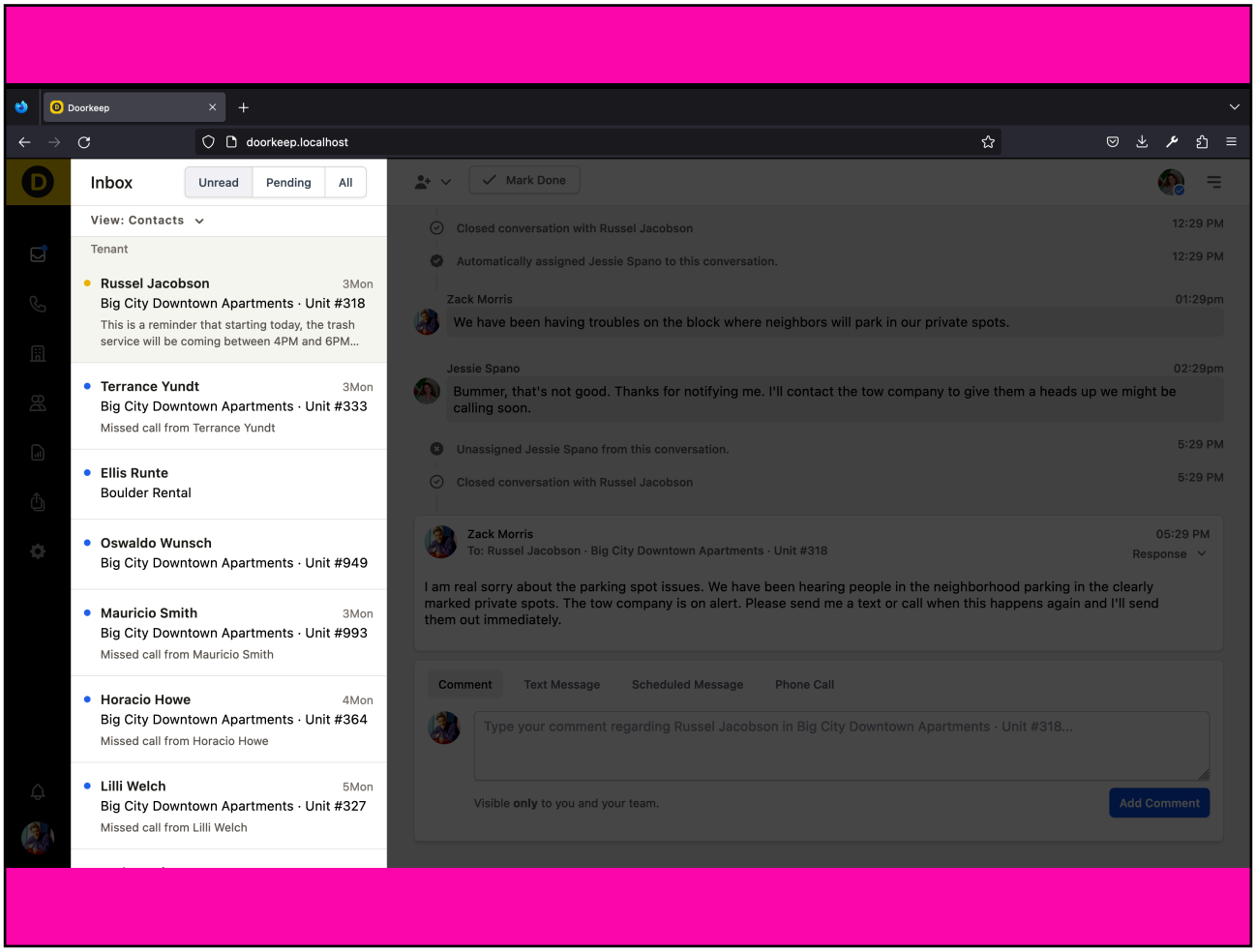


Let's break down the Application frame a bit. There are a lot of subframes in here. Self contained sections. Etc.

Let's start with the "secondary" navigation frame.



Just a reminder of all of the frames again.

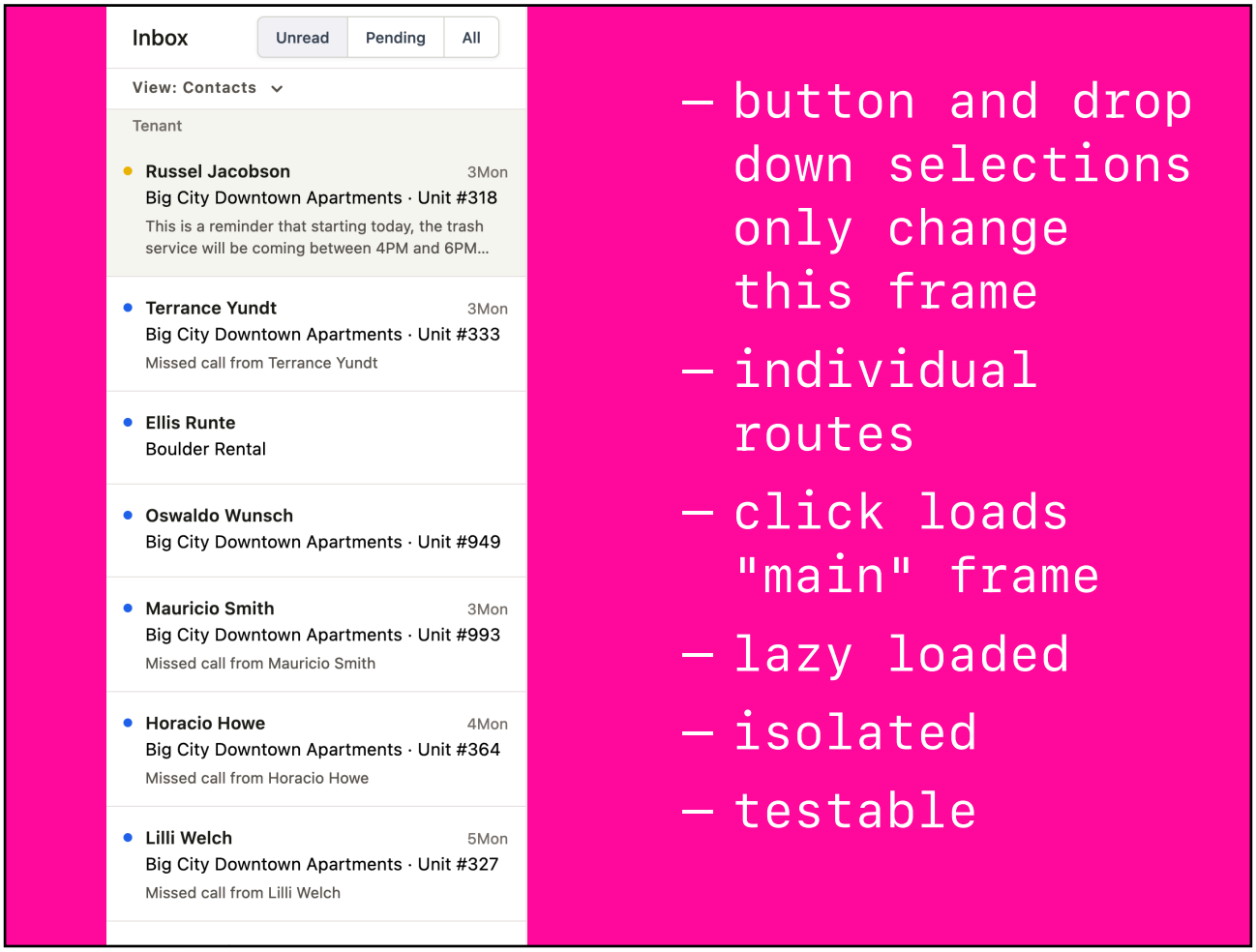


Here is the secondary navigation frame. Let's take a look at it because there is a lot going on. First let's see the frame code that loads the list.

How it's loaded:

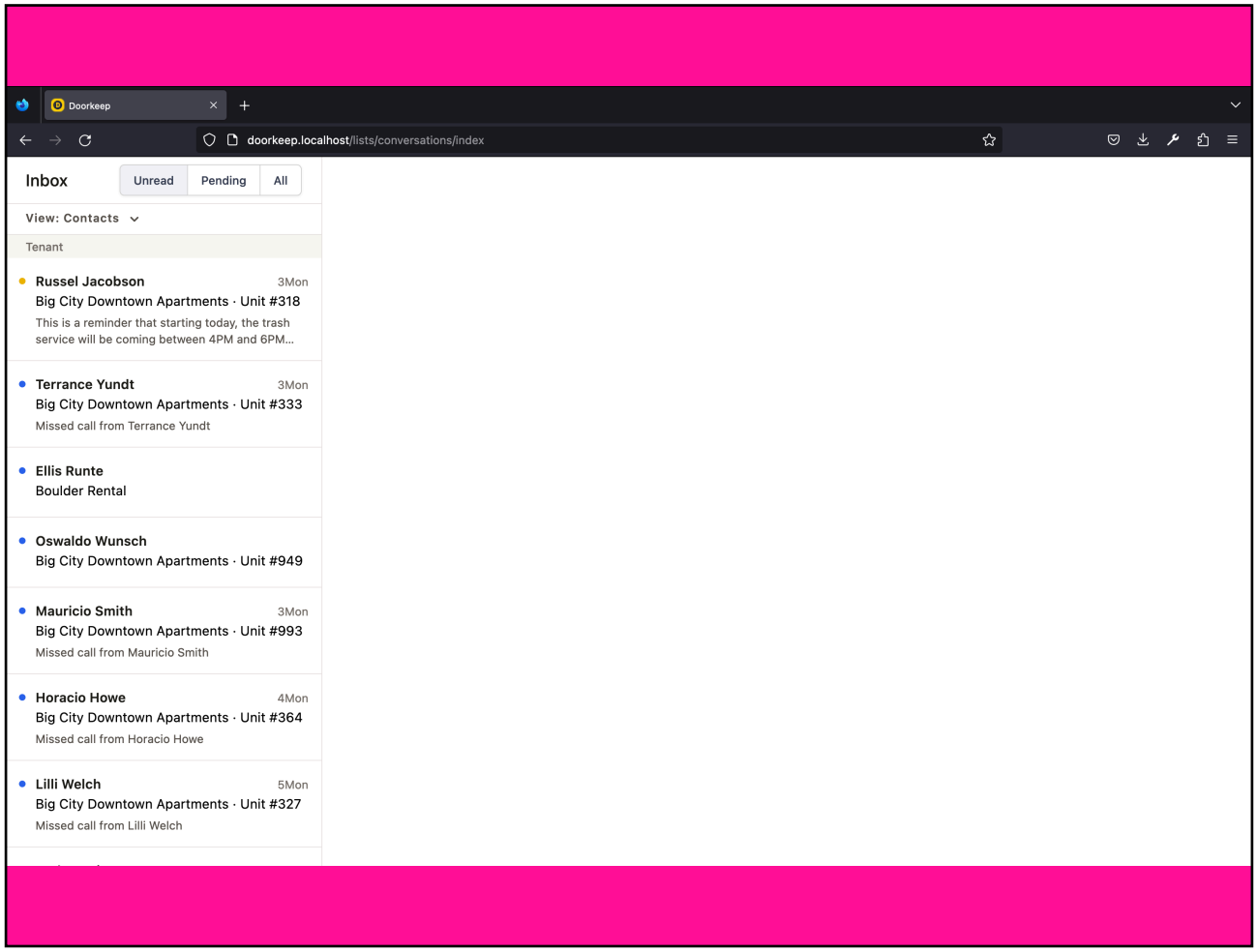
```
<%= turbo_frame_tag "secondary",
src:
lists_conversations_index_path %>
```

I have a separate route for all the secondary navigation. In the Inbox, it needs to load a list of conversations. It'll



Remember this frame is lazy loaded. So it loads on its own via the `src` attribute on the `turbo-frame` tag.

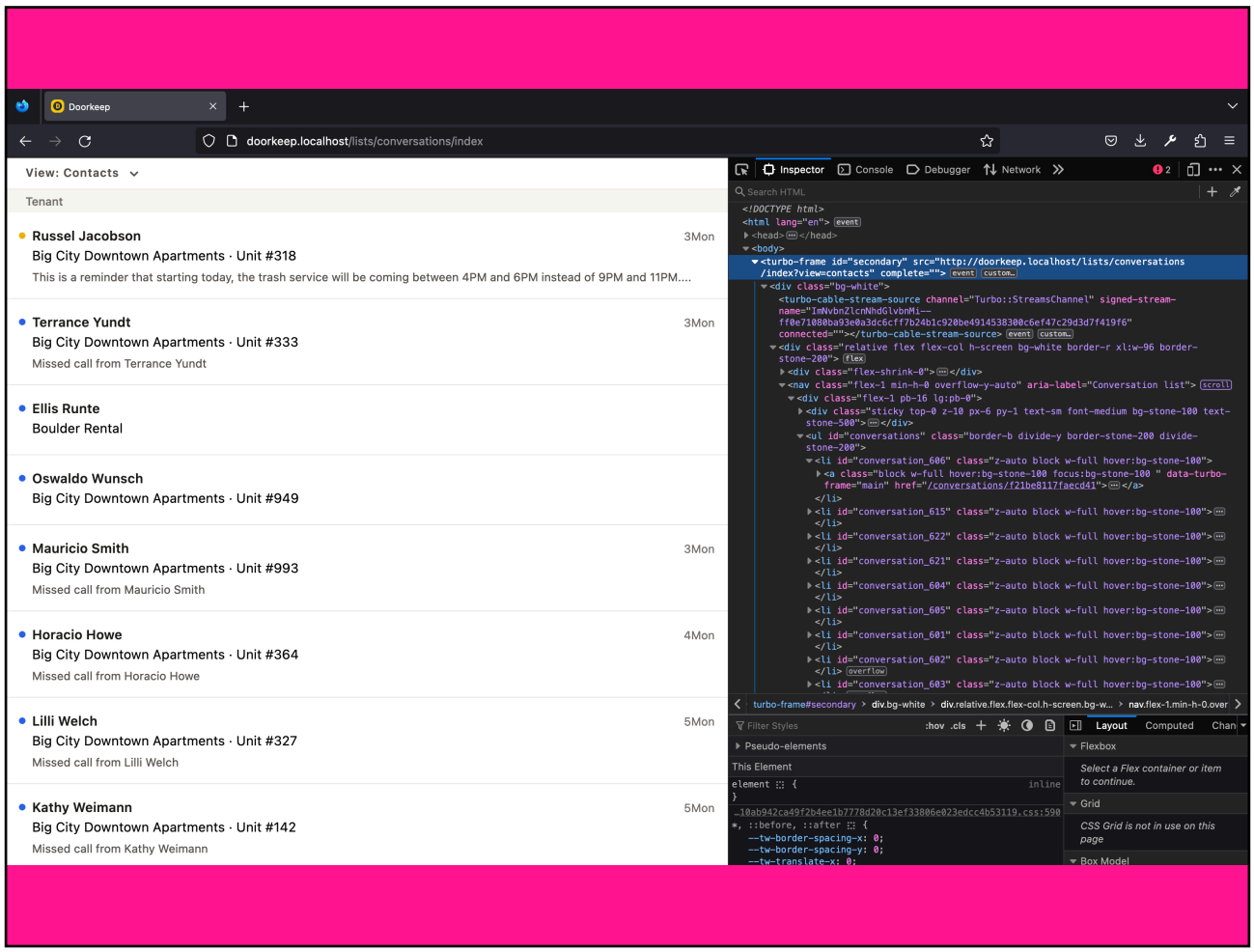
This list being a frame is awesome because all of these drops downs, buttons and such just go back to the route and reload that list with Turbo magic.



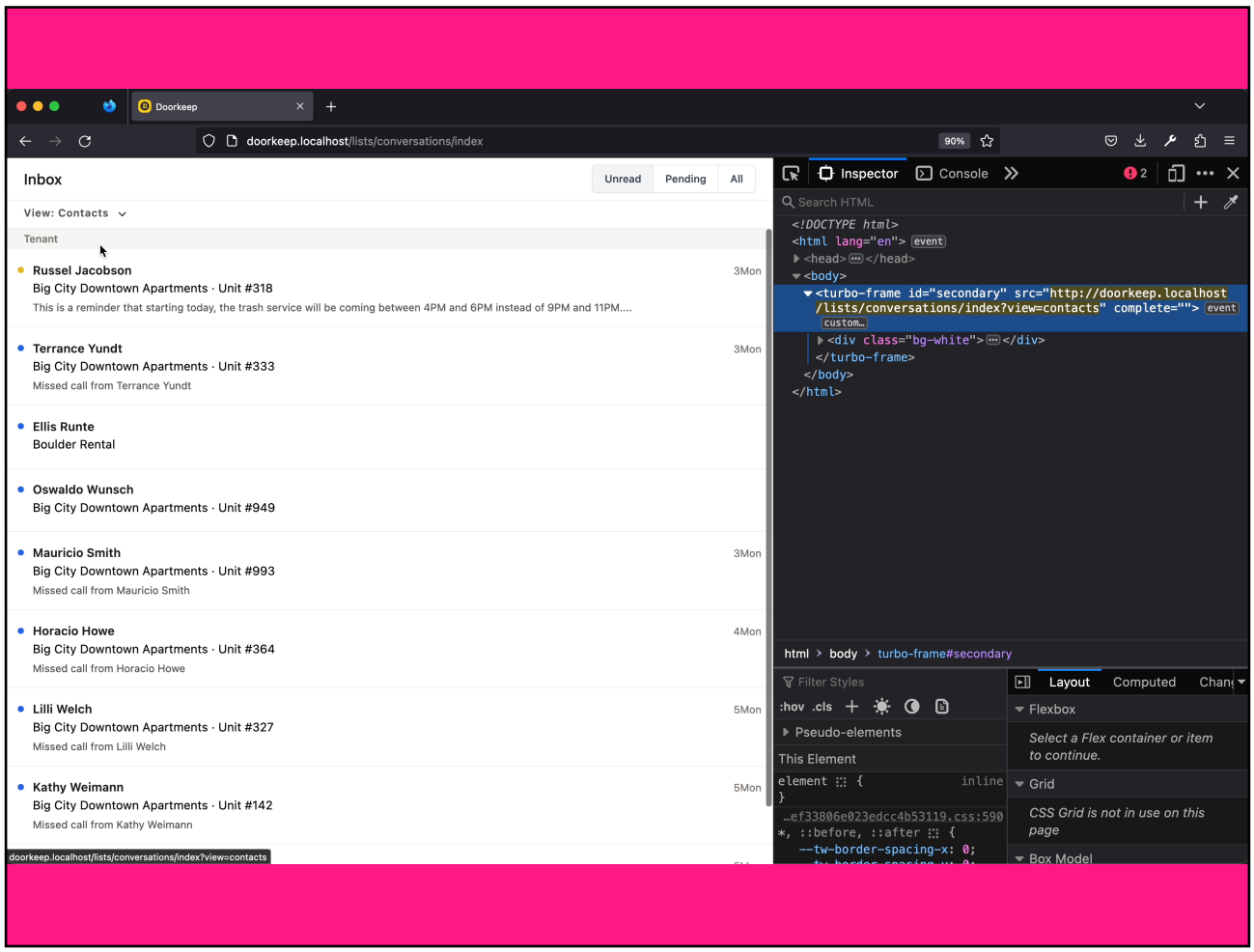
Here is what it looks like if you hit the url for this frame directly.

The extra space and styling is on purpose but it has nothing to do with frames or anything.

This helps with Turbo Native too as far as I can tell. More on this later.



Here is a view in the inspector.



Notice the inspector there where `src` attribute changes? Isn't that cool? It hits one route that its only job is to design a list. The frame encapsulates it, loads it asynchronously and reloads upon change of the button or dropdown without reloading the entire page.

Plus, as you might have thought I can just load this single route to test stuff or design stuff.

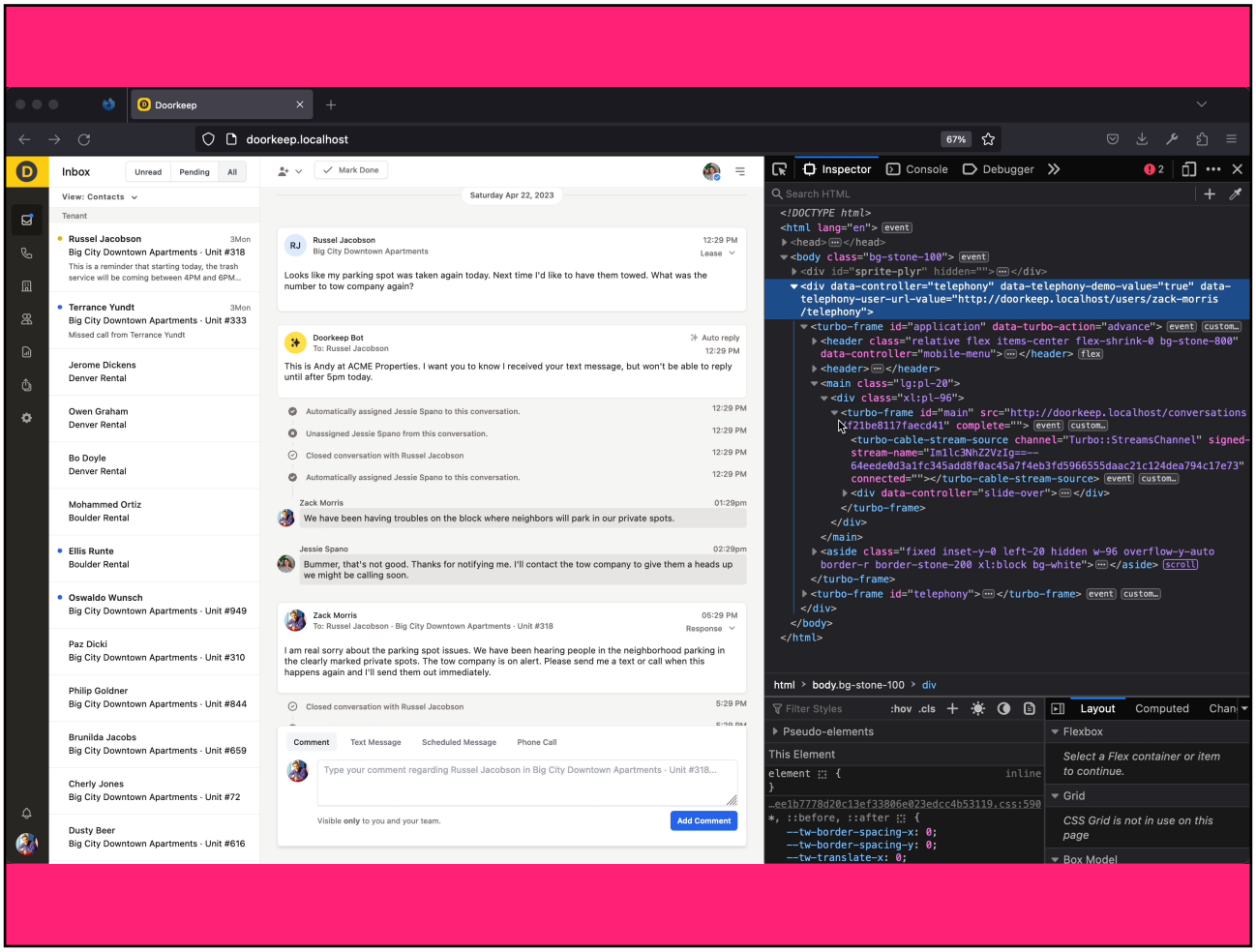
Thinking about the mobile app, Turbo Native opens up some nice to haves too.

```
<%= link_to
conversation_path(conversation),
data: { turbo_frame: "main" } ...
%>
```

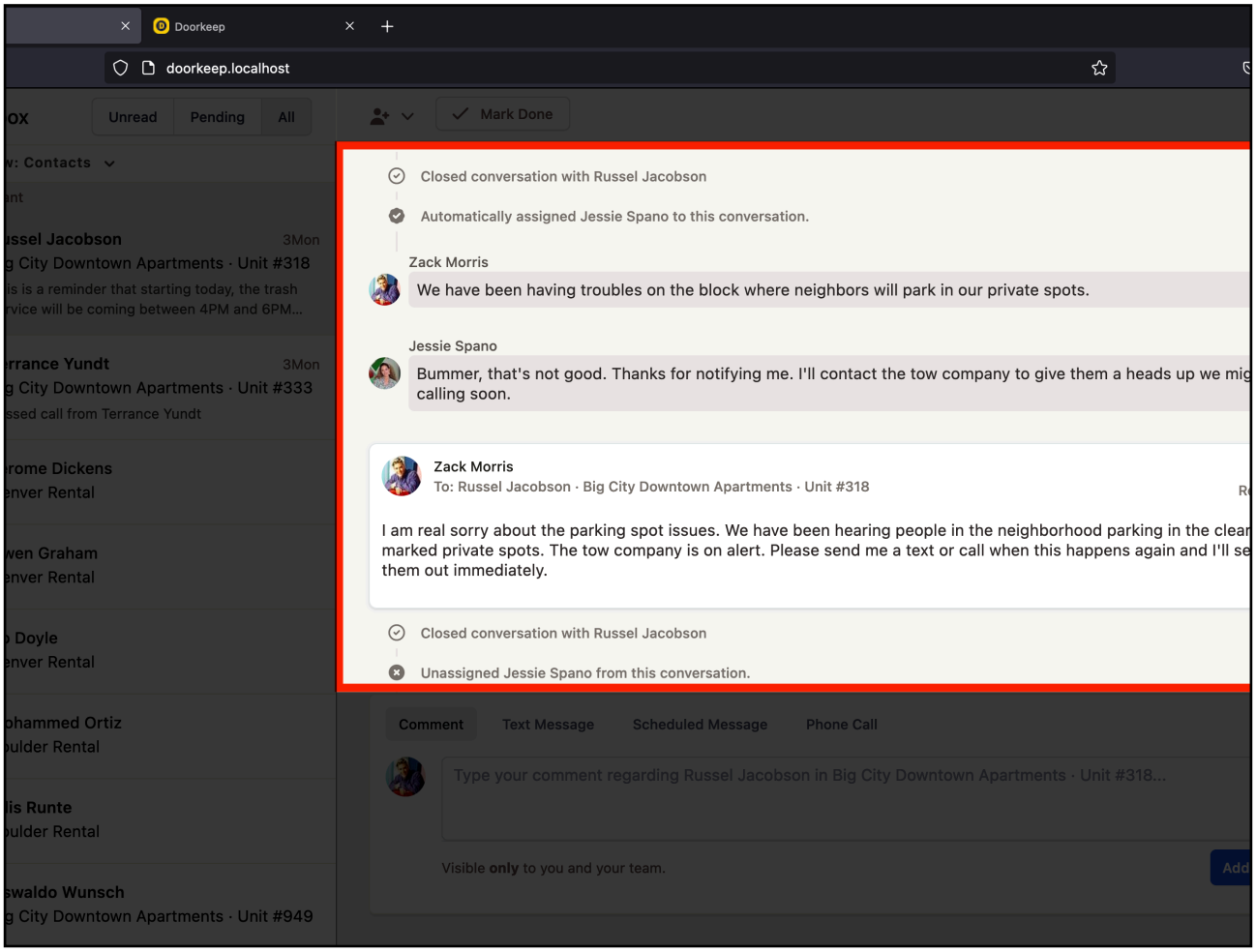
Navigate from one frame, to another.

How do we click in the secondary frame and have it change everything in the "main" frame? By defining the desired turbo frame again.

So when clicking the "main" navigation bar on the far left of the screen, it'll be sure to target the change to the "main" frame.



Here I am clicking around in the secondary frame and it loads the main frame.



This frame actually.

Now that we have this frame loaded from the secondary navigation. Let's talk about how it is loaded.

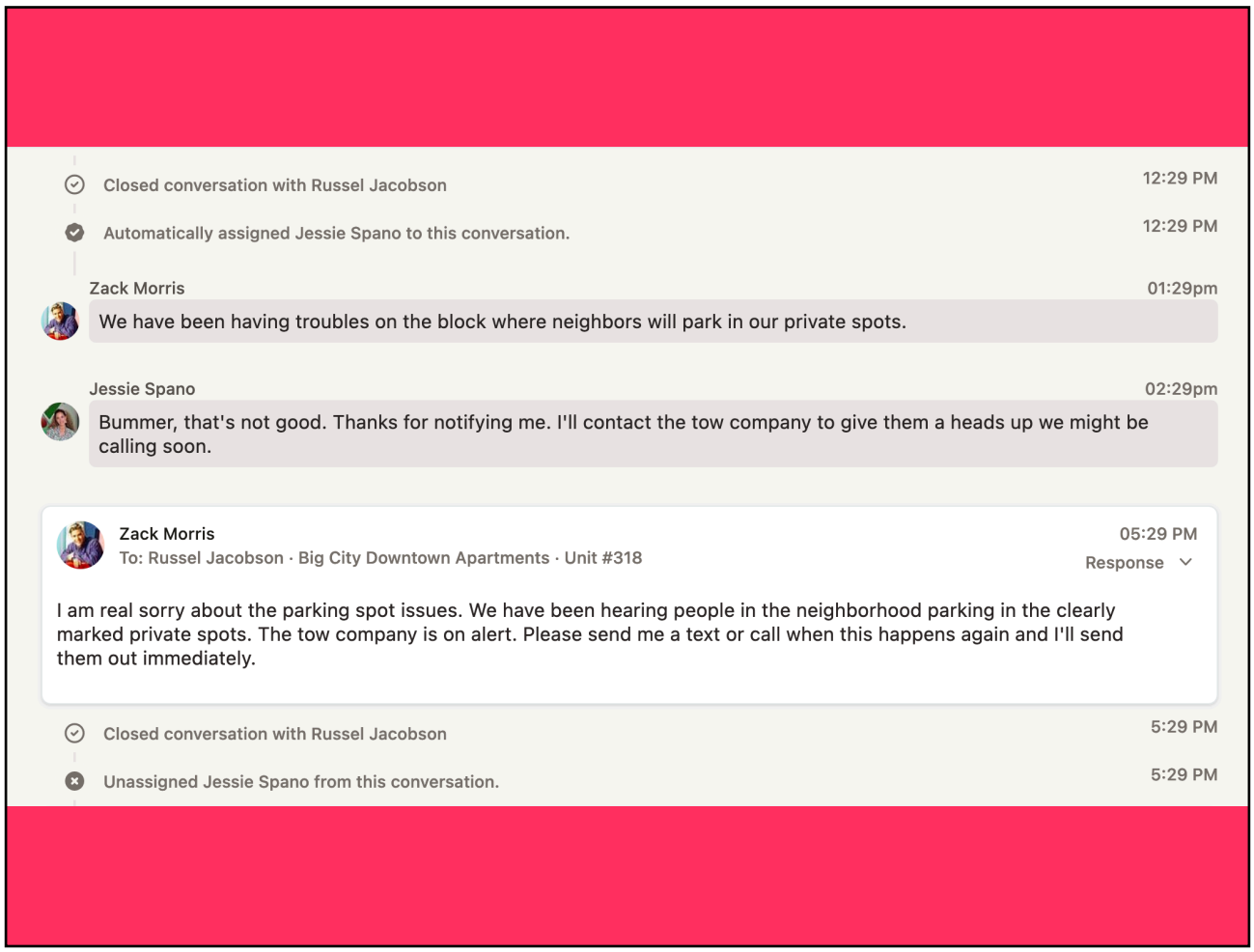
```
turbo_frame_tag "messages", src:
conversation_messages_path(@conversation),
autoscroll: true, loading: "lazy"
```

Lazy load ~~messages~~ activities with autoscroll.

The full message load lazily also. Oh btw, I crossed out messages there because this section is being refactored into activities. Since there is a lot more going on in here than just some messages. That detail was added for context.

Another benefit of frames is autoscroll. So it does the scrolling to the top of the list for me after load.

You can optionally put in `loading: "lazy"` to use an empty loading div.



Not available in this gif, but I have a loading spinner `<div>` that can play by default using. This is commonly done by just putting a place holder (`<div>` or `<img>` or whatever) inside the frame for first load. When the new content is loaded via `src=` that placeholder will be replaced. Sometimes I just leave this blank.

Turbo Frame Quick Tips

Let's do a few quick frame tips...

`target: _top`

`data: { turbo_frame: "_top" }`

Breaking free from your turbo-frame

Remember, any form action or `link_to` action within a frame is going to happen inside that frame.

Also remember, that turbo-frames act like `<iframe>`s do. Sometimes you need to just reload the whole page though.

link_to

```
link_to "Foo", foo_path, target:
"_top"

link_to "Foo", path, data: {
turbo_frame: "_top" }

link_to ..., data: {
turbo_method: :delete,
turbo_confirm: "U sure?" }
```

Reminder about using `data:` and `target:` in your `link_to`.

You will come across "CONTENT MISSING" from time to time. If you see this, something happened. Probably a server error or a template error. You don't get the stack trace with Turbo Frames.

...that's Turbo Frames.

Frames really allowed me to breaking apart the app into smaller controller and templates. Then receive the benefit of a fast performant frontend with very little to no Javascript.

One good approach is to start decomposing your app with frames first before streams. Work your way inward or outward with a form or two. Frames are awesome!

Okay, moving on...

Turbo Streams

How about that screenshot?

Thought process for Turbo Streams.

I reach for *turbo streams* to update the UI outside of the current frame and any other element that a user needs to see as a result of a model change.

Turbo Streams is great for going beyond Frames and updating parts of your app that are more complex or maybe updated in the background, etc.

Or maybe you just want more control. Turbo Stream can do a lot and it's where I started out learning Turbo. Turbo Streams starts off simple, but I can get analysis paralysis on it pretty quick.

I'm going to share just a few examples on how I use streams in Doorkeep...

Example: new comment

Incoming text message or private comment made by a user on the conversation.

Let's see some generalized code first.

Message.rb

```
after_create_commit do
 broadcast_append_to :messages,
target:"#
{dom_id(conversation)}_messages",
partial: "messages/message",
locals: { message: self }
end
```

This is an example on the Message object.

Everything happens as a callback. This is helpful because the message object can be updated in parts of the app that are not the UI.

I'm broadcasting to `:messages` here. Targeting a dom_id of `conversation_ID_messages` and rendering the partial and sending it the Message object.

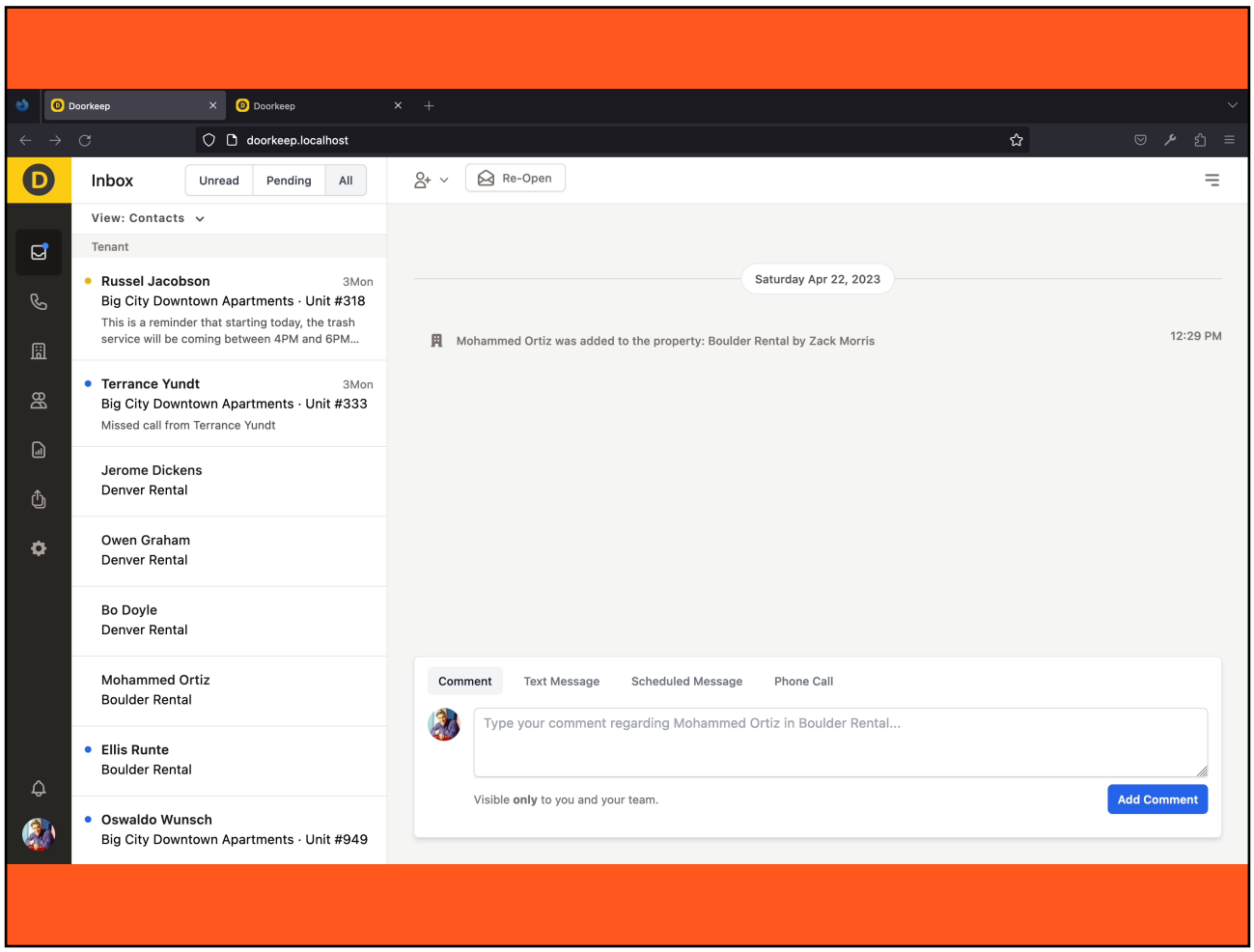
messages.html.erb

```
<%= turbo_stream_from :messages
%>
...
<ul id="<%= dom_id(conversation)
%>_messages"></ul>
```

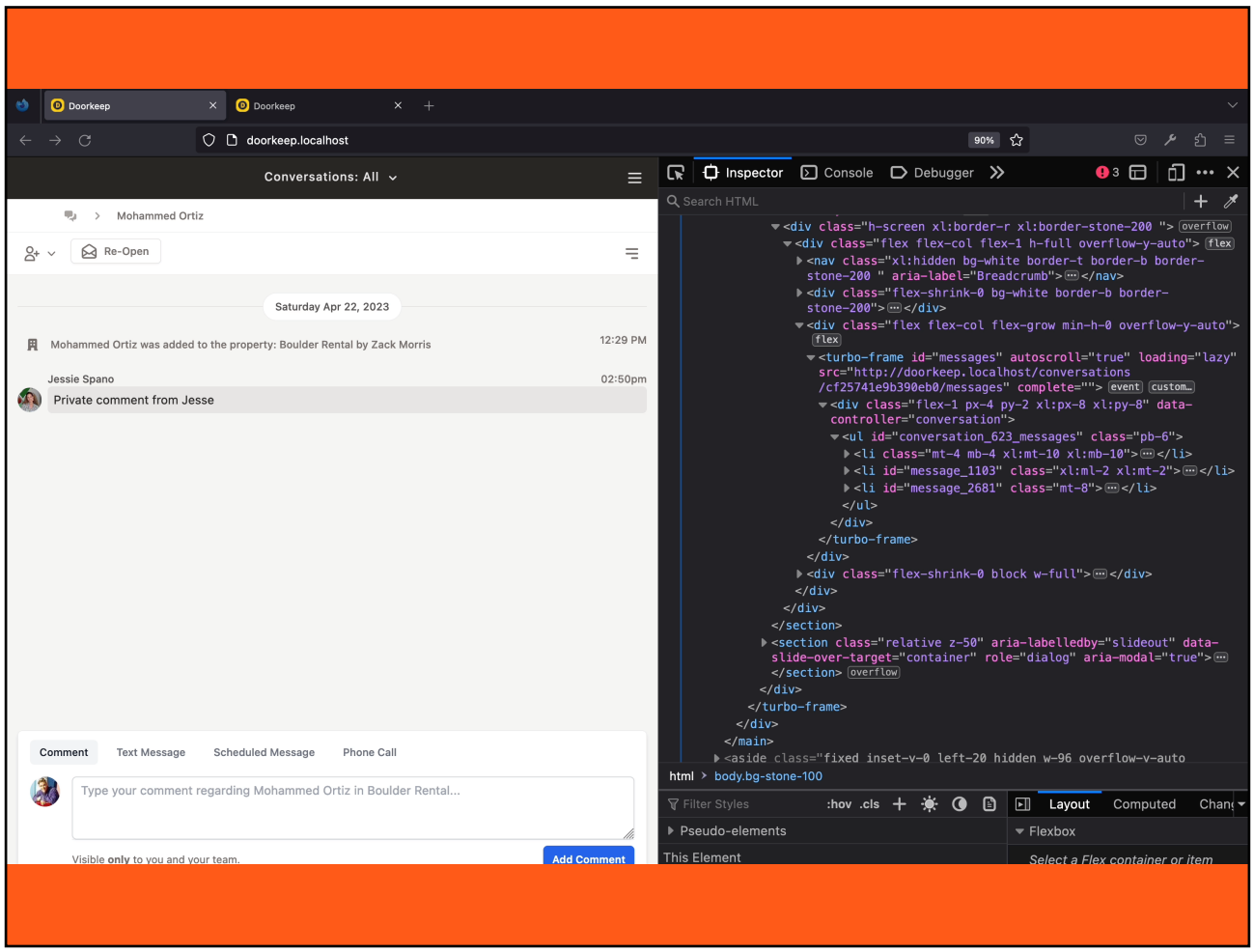
As you can see the stream is setup to receive the broadcast from the model object before.

So now it's listening. When it receives that messages, it'll append a new `message/messages` partial inside the unordered list.

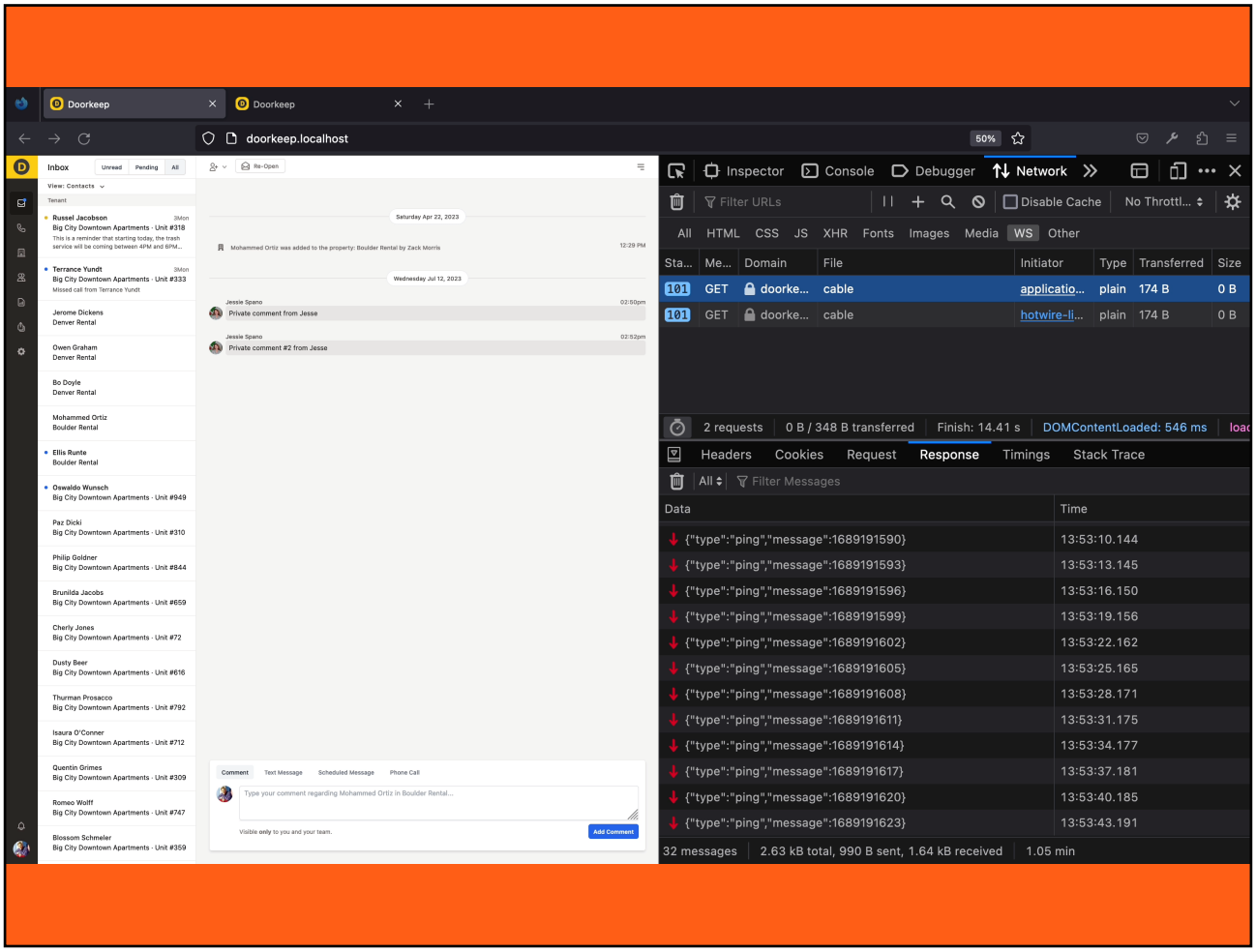
Let's see it in action..



This is a comment coming in from another user logged in via another browser session.



Here is the comment appending itself inside the unordered list.



Here is the websocket stream.

Example 2: Update or removal of a comment

If you add a new comment to the conversation, then you can easily update or delete them too. Let me show you the code I'll skip the gifs this time.

Message.rb

```
after_update_commit do
 broadcast_replace_to :messages,
target: dom_id(self), partial:
"messages/message", locals: {
message: self }
end

  after_destroy_commit do
    broadcast_remove_to :messages
  end
```

This is an update broadcast example on the Message object.

The Update broadcast really doesn't need the additional elements, but I have them because of consistency reasons in other parts of the app.

messages.html.erb

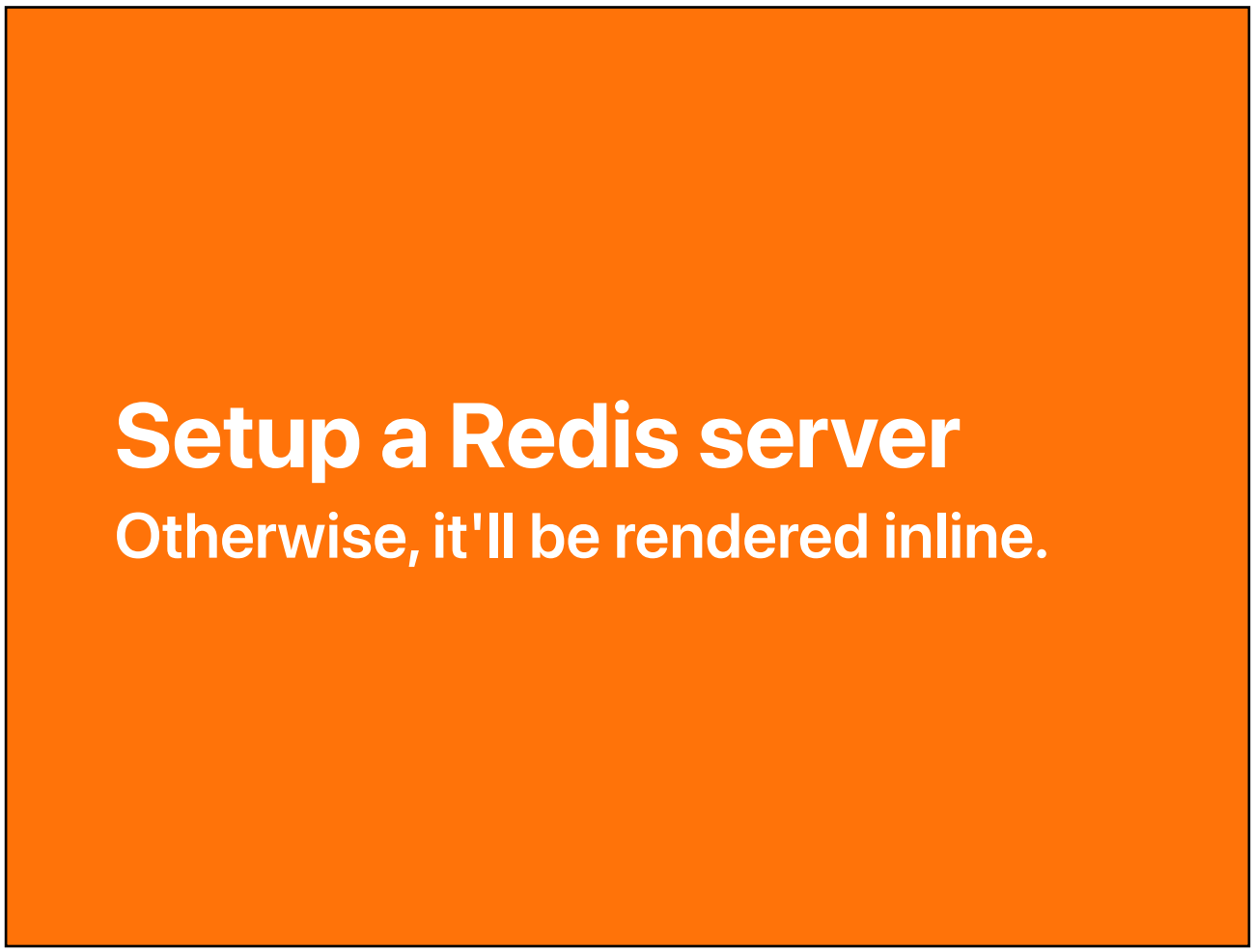
```
<%= turbo_stream_from :messages
%>
...
<li id="<%= dom_id message %>">
</li>
```

The stream is setup to receive the broadcast. It'll use all the ActionCable stuff you probably never used before.

Now it's awesome!

Turbo Stream Tips

How about a few Turbo Stream tips if we have time.



Not a tip, but you need a Redis server.

Perhaps consider multiple Redis servers.

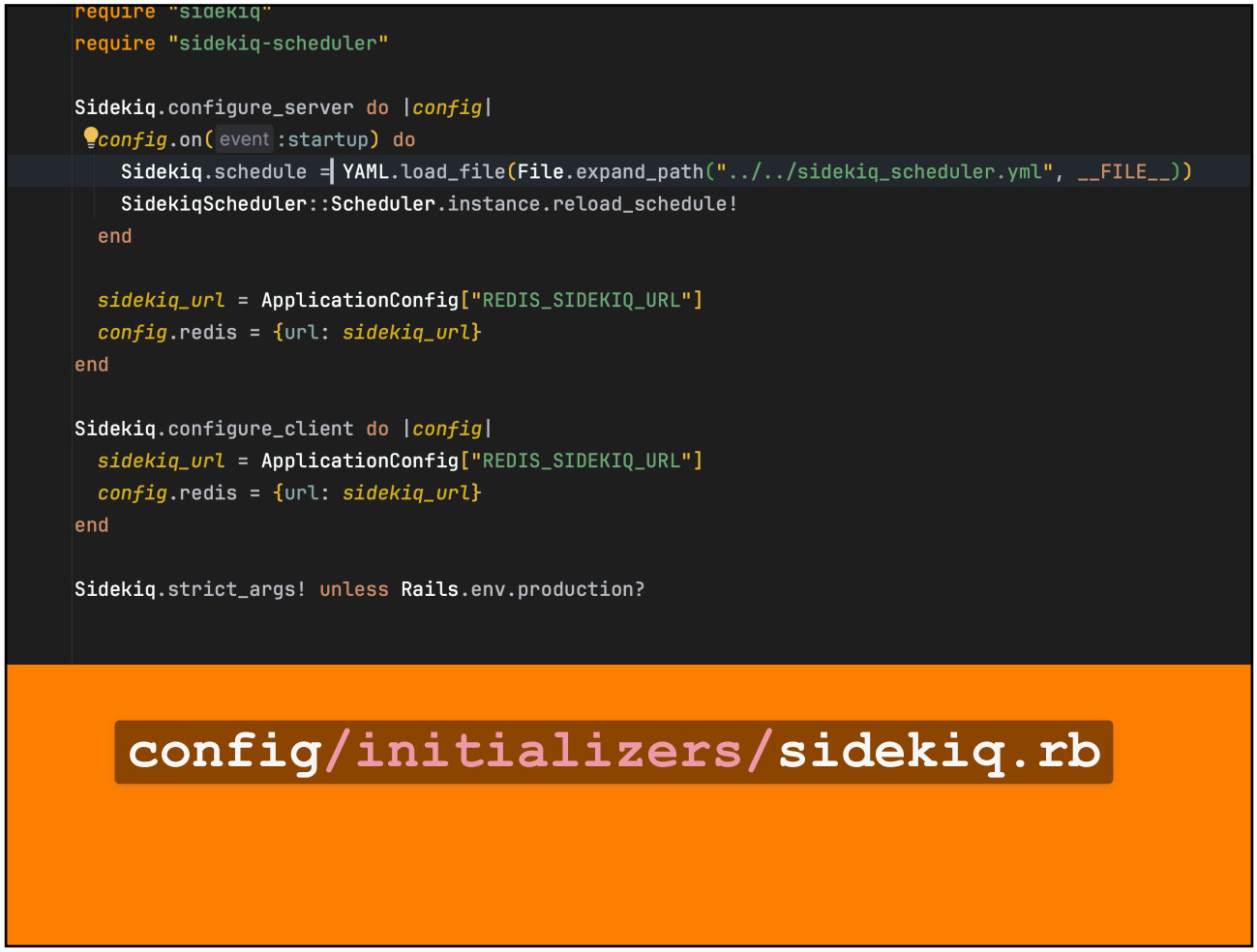
I use one Redis server for Sidekiq and a seperate server for rails caching as well as broadcasting streams on.

I use Redis as a cache store. Since I use it for caching, I needed to be sure to have a different instance for Sidekiq. Cache stores are configured to expire keys after a period of time or when memory is low. Sidekiq uses Redis as a permanent data store. I don't want to be losing jobs due to expiring keys.

REDIS URL Env Variables

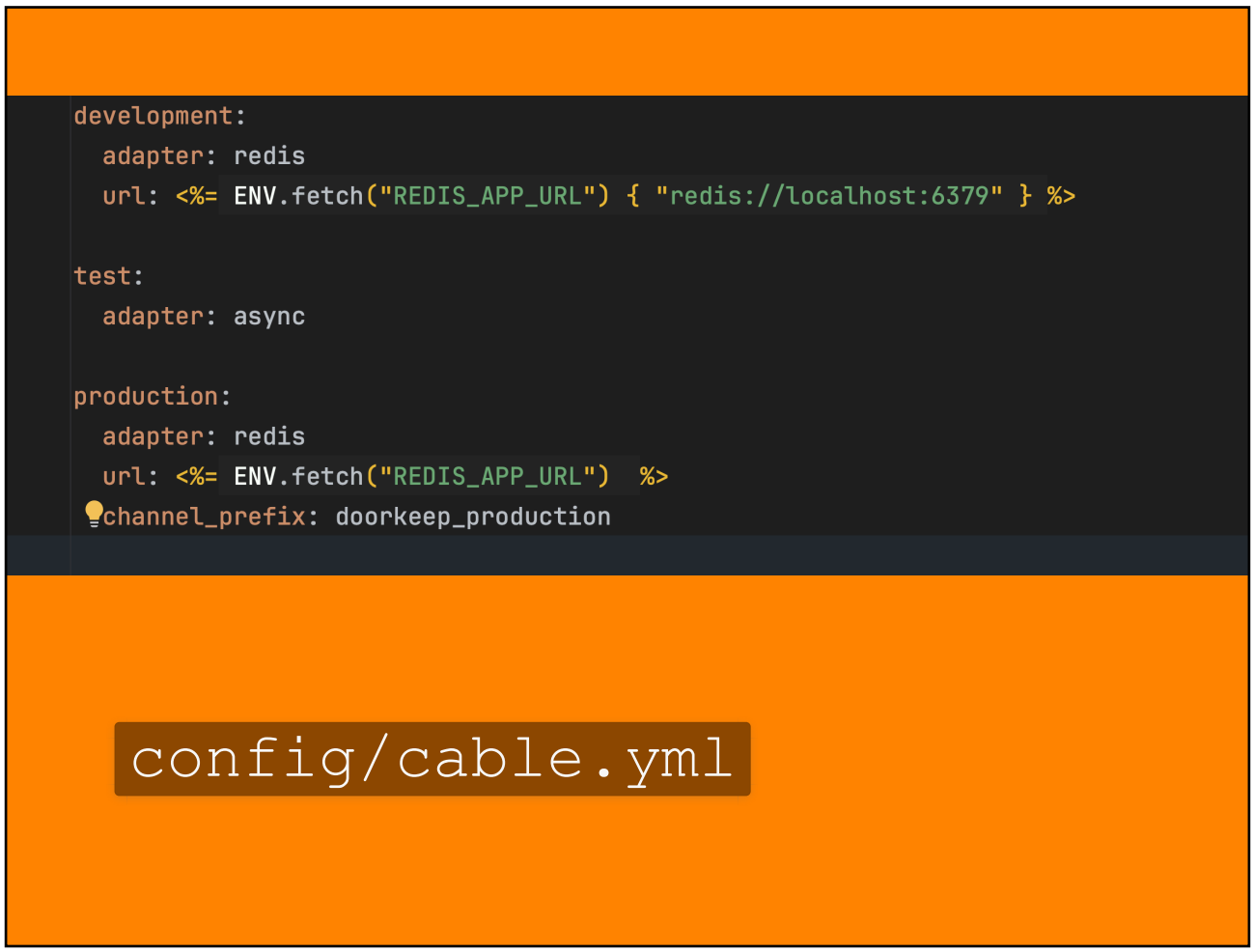
```
REDIS_APP_URL:
redis://localhost:6379/0

REDIS_SIDEKIQ_URL:
redis://localhost:6379/0
```

Two environment variables.



Showing an example of `config/initializers/sidekiq.rb`.

This setups up Sidekiq to use a specific URL. That way Sidekiq uses its very own Redis.



Showing an example of `config/cable.yml`. Keeping streams using a specific Redis server.

Scope objects you are broadcasting to

In Doorkeep, I need everyone under each account who is logged in to see a new messages or conversation updates (for example) in real time; not just one user. But I don't want any outside the account (or team) to see it.

My approach is simple model scoping and some Model access permissions. However you do it, keep this in mind during broadcasts.

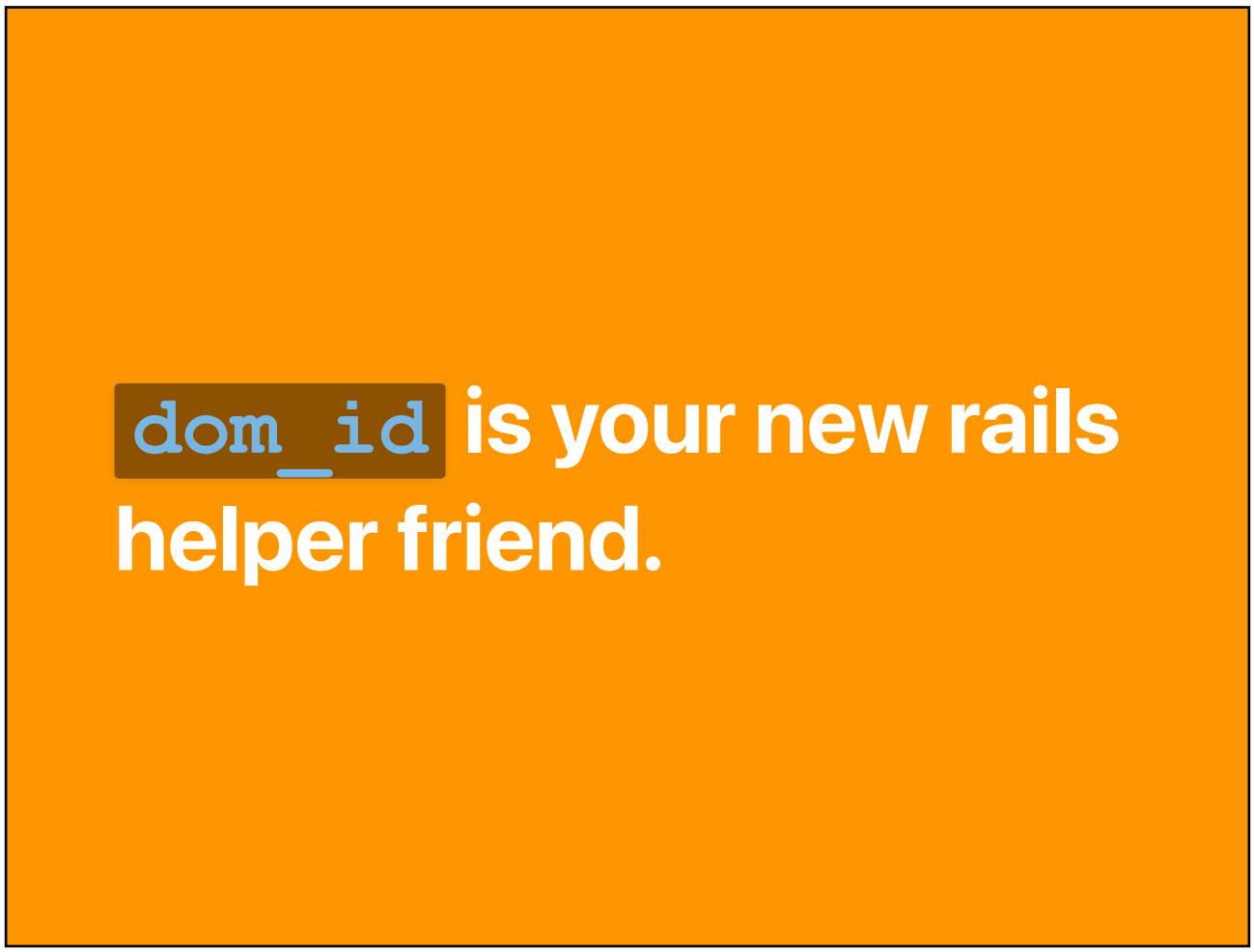
ActiveRecord commit callbacks

`after_create_commit`

`after_update_commit`

`after_destroy_commit`

Use the callbacks that reflect after the object has been property committed to the database.



Just throwing this slide up to say you'll be using it a lot. And not just templates.

Controller broadcasts vs Model?

`broadcast_append_to` (etc)

vs

`create.turbo_stream.erb` (broadcast in the template)

It is easy to get into analysis paralysis on where your broadcasts should originate from. It's also weird still in my head to make changes to the front end from my model. But is it actually weird?

Model broadcasts for objects that *no matter what* will show up in the UI. Ie: A new message that comes in via webhook or a notification that happens from a scheduled job perhaps.

I use controller broadcasts to update parts of the UI that happened from this specific controller action. Sort of think of it as specific cases.

It is still weird to have
`create.turbo_stream.erb`
and
`delete.turbo_stream.erb`
and
`updated.turbo_stream.erb`

I did this when I had view templates for Rails APIs. This isn't foreign. I guess though since these actions are happening still in a browser instead of through magical api based http requests from something else, it can still be not easy to wrap a brain around.

Ultimately Websockets (like a client using an API) cannot follow redirects.

... and that's Turbo Stream.

Turbo Native

Everything we just spent the last many minutes on was to build a fast loading modern web application, but also setup for building native applications with the Turbo Native bridge.

In progress. I still have questions and I want to share more.

As of this very moment, I have an iOS and Android project started with the basics in place. But barely testable. I'm learning more about how my frames will be used as routes are one of the core elements of Turbo Native.

Remember, I and you don't need to wait for Hotwire's Strada to start working with Turbo Native. You simply need to decompose like we have and have a responsive design.

Building a mobile application is whole new can of worms. App Store submissions, etc. I'd really like to demonstrate building a Turbo Native app from start to finish when I have mine in the app stores. So...

Unlike the failed movie of Mac and Me, I really hope to be back soon. Maybe a talk where I can go through everything I learned launching the Turbo Native app.

Anyway, thank you for listening. Have a great night!

Any questions or thoughts or whatever?