Hotwire Cheatsheet for Ruby on Rails Developers

This cheatsheet provides a comprehensive quick reference to Hotwire's core components in Ruby on Rails, including Turbo, Stimulus, and their integration for building fast, dynamic web applications. It covers essential setup steps, code examples, debugging techniques, and best practices to streamline development and improve performance.

Hotwire for Ruby on Rails Developers

Introduction to Hotwire

Hotwire, which stands for HTML Over-The-Wire, is an alternative approach to building modern web applications without using much JavaScript by sending HTML instead of JSON over the wire.

It's designed to work seamlessly with Rails, providing a way to create rich, dynamic user interfaces with significantly less client-side JavaScript.

Hotwire is composed of Turbo and Stimulus. Turbo handles server-rendered HTML updates and Stimulus provides a lightweight JavaScript framework for enhancing the HTML.

Hotwire Architecture

The core idea is to leverage server-side rendering for most of the application logic and use HTML updates to change parts of the page. Here's a simplified diagram:

```
[Browser] <--> [Rails Server]
      HTML          HTML
```

1. **User Interaction**: The user interacts with the web page (e.g., clicks a button, submits a form).
2. **Request to Server**: The browser sends an HTTP request to the Rails server.
3. **Server Processing**: The Rails server processes the request and renders HTML.
4. **HTML Response**: Instead of JSON, the server sends back HTML fragments.
5. **Turbo Updates**: Turbo intercepts the HTML response and intelligently updates the DOM, replacing only the necessary parts of the page.

Benefits of Hotwire

- **Reduced JavaScript**: Dramatically reduces the amount of JavaScript needed, simplifying the front-end.
- **Faster Initial Load**: Server-rendered HTML leads to faster initial page loads and better SEO.
- **Improved Performance**: Sending smaller HTML fragments instead of large JSON payloads improves performance.
- **Enhanced Productivity**: Easier to develop and maintain applications with less complexity.

Turbo Drive

Turbo Drive automatically speeds up links and form submissions by intercepting them and making them into `fetch` requests. It then replaces the `<body>` tag with the response.

To disable Turbo Drive on a specific link:

```
<a href="/path/to/page" data-
turbo="false">Link</a>
```

To disable Turbo Drive entirely:

```
<meta name="turbo-visit-control"
content="reload">
```

It also maintains a persistent history and provides advanced visit control.

Example of using Turbo Drive to navigate:

```
<%= link_to "Go to Home", root_path %>
```

Turbo Frames

Turbo Frames allow you to divide a page into independent sections that can be updated individually.

A Turbo Frame looks like this:

```
<turbo-frame id="my_frame">
  Content to be updated
</turbo-frame>
```

When a frame's content changes, only that frame is updated, not the entire page. This reduces bandwidth and makes for a snappier user experience.

Example of updating a Turbo Frame:

```
# Controller
def update
  @item = Item.find(params[:id])
  @item.update(item_params)
  render turbo_stream:
turbo_stream.replace("item_#{@item.id}",
partial: 'item', locals: { item: @item
})
end
```

Turbo Streams

Turbo Streams deliver page changes as fragments of HTML to be appended, prepended, replaced, updated, or removed.

Example:

```
render turbo_stream:
turbo_stream.append("comments", partial:
"comments/comment", locals: { comment:
@comment })
```

These streams are sent over WebSocket connections, or after form submissions and link clicks.

Another example:

```
render turbo_stream:
turbo_stream.remove("comment_123")
```

Stimulus.js

Stimulus is a modest JavaScript framework for augmenting your HTML. It doesn't take over the entire page; instead, it's designed to work well with server-rendered HTML.

A Stimulus controller looks like this:

```
//
app/javascript/controllers/hello_control
ler.js
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  static targets = [ "name", "output" ]

  greet() {
    this.outputTarget.textContent =
`Hello, ${this.nameTarget.value}!`
  }
}
```

It connects JavaScript objects to elements in your HTML using data attributes.

And the corresponding HTML:

```
<div data-controller="hello">
  <input data-hello-target="name"
type="text">
  <button data-action="click-
>hello#greet">Greet</button>
  <span data-hello-target="output">
</span>
</div>
```

Rails Integration

Rails provides excellent support for Hotwire through the `turbo-rails` gem. This gem adds helpers for generating Turbo Frames, Streams, and more.

Ensure that you include `turbo-rails` in your Gemfile and run `bundle install` to get started.

You can generate a new Rails application with Hotwire pre-configured using the `--turbo` flag:

```
rails new myapp --turbo
```

Practical Examples

Consider a simple to-do list application. With Hotwire, adding a new to-do item involves submitting a form, the server renders the new item as HTML, and Turbo Streams append it to the list without a full page reload.

This is much simpler than implementing the same functionality with a traditional JavaScript framework, which would require writing API endpoints, handling JSON responses, and manually updating the DOM.

Another common use case is real-time updates. For example, a chat application can use Turbo Streams over WebSockets to push new messages to all connected clients in real time. The server renders the message as HTML, and Turbo Streams ensure it's instantly displayed in the chat window.

Hotwire Resources

Hotwire Official Website - Official documentation and guides for Hotwire.

Turbo Handbook - Comprehensive guide to Turbo.

Stimulus Handbook - Comprehensive guide to Stimulus.

Hotwire Source Code (GitHub) - Source code for Turbo and Stimulus.

GoRails Hotwire Tutorials - A collection of Hotwire tutorials on GoRails.

Drifting Ruby Hotwire Episodes - Hotwire tutorials and screencasts.

Hotwire Newsletter - Stay updated with the latest news and updates from the Hotwire community.

Stimulus Components - A curated collection of reusable Stimulus components.

Masilotti.com Newsletter - Insights and articles on modern web development, including Hotwire.

Turbo Overview

Turbo Fundamentals

Turbo is a suite of techniques for building modern web applications by sending HTML over the wire. It enhances Rails apps by providing a way to update parts of a page without full page reloads, resulting in faster and more responsive user experiences.

Turbo consists of:

- **Turbo Drive:** Automatically makes links and forms faster using AJAX.
- **Turbo Frames:** Decompose complex pages into independent components.
- **Turbo Streams:** Deliver page changes over WebSocket, SSE, or as a result of form submissions.

Turbo Drive

Turbo Drive intercepts all clicks on `<a>` tags and form submissions. It issues AJAX requests, expects HTML in response, and replaces the `<body>` of the current page with the `<body>` of the response.

This makes navigation feel instantaneous.

How it works:

1. You click a link.
2. Turbo Drive prevents the browser from following the link.
3. Turbo Drive issues an AJAX request for the page.
4. When the AJAX request completes, Turbo Drive replaces the current `<body>` with the `<body>` from the response.
5. Turbo Drive merges the `<head>`.

Example:

```
<a href="/articles/1">Show Article</a>
```

Clicking this link triggers a Turbo Drive request.

To disable Turbo Drive on a specific link, use the `data-turbo="false"` attribute:

```
<a href="/articles/1" data-turbo="false">Show Article</a>
```

Turbo Frames

Turbo Frames allow you to break a page into independent sections that can be updated individually.

Each frame is defined by a `<turbo-frame>` tag with a unique `id`.

Example:

```
<turbo-frame id="article_1">
  <h2>Article Title</h2>
  <p>Article content...</p>
</turbo-frame>
```

When a link or form inside a Turbo Frame is clicked, Turbo Drive only updates the content within that frame.

This isolates updates and improves perceived performance.

Lazy Loading with `src`:

```
<turbo-frame id="comments"
src="/comments"></turbo-frame>
```

This loads the comments section only when the frame is visible, improving initial page load time.

Turbo Streams

Turbo Streams deliver targeted updates to the page via WebSocket, SSE (Server-Sent Events), or as a result of form submissions.

They consist of actions (like `append`, `prepend`, `replace`, `update`, `remove`) that target specific DOM elements.

Example:

```
# app/views/comments/create.turbo_stream.erb

<%= turbo_stream.append "comments", partial: "comments/comment", locals: { comment: @comment } %>
```

This appends a new comment to the element with id `comments`.

To broadcast Turbo Streams from a model callback:

```
# app/models/comment.rb
after_create_commit { broadcast_append_to :comments }
```

Available Actions: (and more)

- `append`
- `prepend`
- `replace`
- `update`
- `remove`

To handle Turbo Stream responses in JavaScript:

```
document.addEventListener("turbo:before-stream-render", (event) => {
  const { target, template } = event.detail
  // Perform custom logic here
});
```

Installation and Setup

1. **Add `turbo-rails` to your Gemfile:**

   ```
   gem 'turbo-rails'
   ```

2. **Run bundle install:**

   ```
   bundle install
   ```

3. **Install Turbo:**

   ```
   rails turbo:install
   ```

   This generates the necessary JavaScript files and modifies your application layout.

4. **Include Turbo JavaScript in your `application.js`:**

   ```
   import "@hotwired/turbo-rails"
   ```

Gotchas and Considerations

JavaScript Execution: Turbo Drive preserves JavaScript state across page visits. Ensure your JavaScript is idempotent and handles Turbo Drive events appropriately (e.g., `turbo:load`, `turbo:frame-load`).

Form Submissions: Ensure your forms are set up to handle Turbo Drive. Use `form_with` in Rails, which generates forms compatible with Turbo.

SEO: Turbo enhances user experience without negatively impacting SEO, as the server still renders full HTML pages.

Accessibility: Ensure your Turbo-powered application remains accessible by using semantic HTML and ARIA attributes.

Testing: Test your Turbo-powered features using system tests to ensure they function as expected in a browser environment.

Turbo Drive

Turbo Drive Overview

Turbo Drive enhances web application speed by intercepting clicks on links and form submissions. Instead of full page reloads, it uses `XMLHttpRequest` to fetch the new page, then updates the current page's `<body>` and `<title>` with the new content. This leads to faster navigation and a smoother user experience.

When a Turbo Drive-enabled link is clicked:

1. Turbo Drive prevents the browser from following the link.
2. It issues an `XMLHttpRequest` to fetch the page.
3. Upon completion, it replaces the current `<body>` with the `<body>` of the response.
4. Merges the content of `<head>` (title, meta tags, etc.).
5. The browser's history is updated.

Enabling Turbo Drive

Turbo Drive is enabled by default in Rails 7 when using the `turbo-rails` gem.
To ensure it's active, verify that `turbo-rails` is in your `Gemfile` and that you've included `//= require turbo` in your `application.js` or equivalent entrypoint.

To explicitly enable Turbo Drive, include the Turbo JavaScript file in your application's JavaScript bundle. For example, using webpacker or jsbundling-rails, ensure the following is present in your `application.js`:

```
import * as Turbo from "@hotwired/turbo"
Turbo.start()
```

Disabling Turbo Drive

You can disable Turbo Drive on specific links or forms by adding the `data-turbo="false"` attribute. This tells Turbo Drive to ignore the link or form and allow the browser to handle it normally (full page reload).

Example (Link):

```
<%= link_to "Full Reload", some_path,
data: { turbo: false } %>
```

Example (Form):

```
<%= form_with url: some_path, data: {
turbo: false } do |form| %>
  <%= form.submit "Full Reload Submit"
%>
<% end %>
```

To disable Turbo Drive application-wide (not generally recommended), you can remove the `turbo-rails` gem or prevent the Turbo JavaScript from loading. However, it's usually better to selectively disable Turbo Drive where necessary.

Turbo Drive Events

`turbo:before-visit`	
	Fired before Turbo Drive visits a location. Can be used to prevent the visit by calling `event.preventDefault()`.
`turbo:visit`	
	Fired when Turbo Drive starts a visit.
`turbo:before-cache`	
	Fired before Turbo Drive caches the page. Useful for cleaning up temporary resources.
`turbo:before-render`	
	Fired before Turbo Drive renders the new page. Allows modification of the new `document.body` before it replaces the current one.
`turbo:render`	
	Fired after Turbo Drive renders the new page.
`turbo:load`	
	Fired after Turbo Drive finishes loading the new page. Similar to `DOMContentLoaded`.
`turbo:frame-render`	
	Fired after a Turbo Frame is rendered.

Handling JavaScript with Turbo Drive

Since Turbo Drive replaces the `<body>` of the page, you need to ensure your JavaScript is properly re-evaluated on each Turbo Drive visit. Use the `turbo:load` event to initialize JavaScript that needs to run after each page load. Avoid using `DOMContentLoaded` directly, as it only fires on the initial page load.

Example:

```
document.addEventListener('turbo:load',
() => {
  console.log('Turbo Drive page
loaded');
  // Initialize your JavaScript here
});
```

For libraries or components that need to be properly disposed of when navigating away from a page, use the `turbo:before-cache` event to clean up resources and prevent memory leaks.

Example:

```
document.addEventListener('turbo:before-
cache', () => {
  console.log('Cleaning up before
caching');
  // Dispose of resources here
});
```

Troubleshooting Turbo Drive

If you encounter issues with Turbo Drive, such as JavaScript not running or unexpected behavior, check the following:

- Ensure `turbo-rails` gem is correctly installed and `turbo` is required in your `application.js`.
- Verify that you're using the `turbo:load` event for initializing JavaScript instead of `DOMContentLoaded`.
- Check for JavaScript errors in the browser console that might be preventing Turbo Drive from functioning correctly.
- Use `data-turbo="false"` to selectively disable Turbo Drive for problematic links or forms to isolate issues.

Inspect network requests in your browser's developer tools to confirm that Turbo Drive is indeed making `XMLHttpRequest` requests instead of full page reloads.
Look for the `Turbo-Visit` header in the request to confirm Turbo Drive is active.

Turbo Frames

Turbo Frames Overview

Turbo Frames allow you to update specific parts of a page without requiring a full page reload. This enhances user experience by making updates feel faster and more responsive.

They work by wrapping sections of your page in `<turbo-frame>` tags. When a link or form inside a frame is activated, Turbo Drive intercepts the request and updates only the contents of that frame, leaving the rest of the page untouched.

This approach reduces server load and network bandwidth, leading to improved performance, especially in complex web applications.

Basic Implementation

HTML (View)

```
<turbo-frame id="my_frame">
  <%= render "partial" %>
</turbo-frame>
```

Rails (Partial - `_partial.html.erb`)

```
<p>Current time: <%= Time.now %></p>
<%= link_to "Refresh", root_path %>
```

Explanation

The `turbo-frame` with `id="my_frame"` will only update when the "Refresh" link is clicked. The rest of the page remains unchanged.

Lazy Loading with Turbo Frames

HTML (Lazy Loading)

```
<turbo-frame id="lazy_frame"
src="/lazy_content">
  <p>Loading...</p>
</turbo-frame>
```

Rails (Controller)

```
#
app/controllers/application_controller.r
b
def lazy_content
  render turbo_frame: "lazy_frame" do
    render partial: "lazy_partial"
  end
end
```

Rails (Partial - `_lazy_partial.html.erb`)

```
<p>This content was loaded lazily!</p>
```

Explanation

The frame with `src` attribute will load content from `/lazy_content` upon insertion into the DOM. A 'Loading...' message is displayed until loaded.

Targeting Turbo Frames

Targeting a specific frame

You can target a specific Turbo Frame from a link or form using the `data-turbo-frame` attribute.

```
<%= link_to "Update Other Frame",
other_path, data: { turbo_frame:
"other_frame" } %>
```

This will update the frame with the ID `other_frame` when the link is clicked.

Targeting `_top`

Using `data-turbo-frame="_top"` will cause a full page reload.

```
<%= link_to "Full Reload", root_path,
data: { turbo_frame: "_top" } %>
```

Targeting `_parent`

Using `data-turbo-frame="_parent"` will target the parent frame of the current element.

```
<%= link_to "Update Parent Frame",
parent_path, data: { turbo_frame:
"_parent" } %>
```

Form Submissions inside Turbo Frames

HTML (Form)

```
<turbo-frame id="form_frame">
  <%= form_with url: "/submit_form",
data: { turbo_frame: "form_frame" } do
|form| %>
    <%= form.text_field :name %>
    <%= form.submit "Submit" %>
  <% end %>
</turbo-frame>
```

Rails (Controller)

```
#
app/controllers/application_controller.r
b
def submit_form
  # Process form data
  render turbo_frame: "form_frame" do
    render partial: "form_result"
  end
end
```

Rails (Partial - `_form_result.html.erb`)

```
<p>Form submitted successfully!</p>
```

Explanation

Submitting the form will replace the content within `form_frame` with the `_form_result` partial. `data: { turbo_frame: "form_frame" }` ensures the response targets the correct frame.

Turbo Streams

Introduction to Turbo Streams

Turbo Streams deliver asynchronous updates to the browser by appending, prepending, replacing, updating, or removing elements on a page. They are typically used with Action Cable for real-time updates but can also be triggered by standard controller actions.

Turbo Streams use specific MIME types (`text/vnd.turbo-stream.html`) to trigger updates. The server responds with HTML fragments that contain instructions on how to modify the DOM.

Turbo Streams are an efficient way to update parts of a page without requiring a full page reload, enhancing the user experience.

They are especially useful for features like live comments, chat applications, and real-time dashboards.

Turbo Streams leverage the `turbo_stream` tag helper and Action Cable's broadcasting capabilities for seamless integration.

Turbo Stream Actions

`append`

Inserts content at the end of the target element.

Example:

```
<%= turbo_stream.append 'comments',
partial: 'comments/comment', locals: {
comment: @comment } %>
```

`prepend`

Inserts content at the beginning of the target element.

Example:

```
<%= turbo_stream.prepend 'comments',
partial: 'comments/comment', locals: {
comment: @comment } %>
```

`replace`

Replaces the entire target element with the new content.

Example:

```
<%= turbo_stream.replace 'comment_1',
partial: 'comments/comment', locals: {
comment: @comment } %>
```

`update`

Updates the content inside the target element, leaving the element itself intact.

Example:

```
<%= turbo_stream.update 'comment_1',
partial: 'comments/comment', locals: {
comment: @comment } %>
```

`remove`

Removes the target element from the DOM.

Example:

```
<%= turbo_stream.remove 'comment_1' %>
```

Creating Turbo Stream Responses

In your controller, you can respond with Turbo Stream templates to trigger updates.

Use the `respond_to` block to handle the `turbo_stream` format.

Example:

```
class CommentsController <
ApplicationController
  def create
    @comment =
Comment.new(comment_params)
    if @comment.save
      respond_to do |format|
        format.turbo_stream { render
turbo_stream:
turbo_stream.append('comments', partial:
'comments/comment', locals: { comment:
@comment }) }
        format.html { redirect_to
@comment.post }
      end
    else
      # Handle errors
    end
  end
end
```

Alternatively, use `turbo_frame_tag` for a more concise approach when dealing with specific frames.

Broadcasting with Action Cable

Action Cable can broadcast Turbo Streams to update multiple clients in real-time.

Use `turbo_stream_from` in your views to subscribe to a stream.

Example:

```
<%= turbo_stream_from 'comments' %>
```

In your model or controller, use `broadcast_append_to`, `broadcast_prepend_to`, `broadcast_replace_to`, `broadcast_update_to`, or `broadcast_remove_to`.

Example:

```
class Comment < ApplicationRecord
  after_create_commit {
broadcast_append_to 'comments', partial:
'comments/comment', locals: { comment:
self } }
  after_update_commit {
broadcast_replace_to 'comments',
partial: 'comments/comment', locals: {
comment: self } }
  after_destroy_commit {
broadcast_remove_to 'comments', target:
"comment_#{id}" }
end
```

Ensure Action Cable is properly configured in your Rails application, including setting up the necessary routes and connection class.

Turbo Stream Templates

Turbo Stream templates are `.turbo_stream.erb` files that define the actions to be performed on the DOM.

These templates are rendered and sent to the client, where Turbo Drive processes them.

Example: `create.turbo_stream.erb`

```
<%= turbo_stream.append 'comments' do %>
  <%= render 'comments/comment',
comment: @comment %>
<% end %>
```

Use partials to keep your templates DRY and maintainable.

Leverage the `turbo_stream` tag helper for concise and readable templates.

Targeting Elements

Using IDs	Target specific elements using their IDs. This is the most common and straightforward approach. **Example:** `<%= turbo_stream.replace 'comment_1', 'New content' %>`
Using CSS Selectors	You can use CSS selectors to target elements, providing more flexibility. **Example:** `<%= turbo_stream.update '.comment', 'Updated content' %>`
Considerations	Ensure your target elements have unique IDs to avoid unintended updates. When using CSS selectors, be specific to prevent broad changes.

Advanced Turbo Streams Usage

Combining Multiple Actions: You can combine multiple Turbo Stream actions in a single response.

Example:

```
<%= turbo_stream.append('comments',
partial: 'comments/comment', locals: {
comment: @comment })
       concat
turbo_stream.update('comment_count',
Comment.count) %>
```

Conditional Updates: Implement conditional logic to determine whether to send a Turbo Stream based on specific conditions.

Example:

```
if @comment.approved?
  render turbo_stream:
turbo_stream.append('approved_comments',
partial: 'comments/comment', locals: {
comment: @comment })
else
  # Do something else
end
```

Custom Actions: While less common, you can define custom Turbo Stream actions by extending Turbo Native.

Error Handling: Implement error handling to gracefully manage situations where Turbo Stream updates fail.

Custom Turbo Stream Actions

Custom Turbo Stream Actions

Backend (Ruby on Rails)

Define helper methods in `app/helpers/turbo_stream_actions_helper.rb` to generate custom Turbo Stream actions.

```
module TurboStreamActionsHelper
  def close_modal
    turbo_stream_action_tag(:close_modal)
  end

  def open_modal(modal_id)
    turbo_stream_action_tag(:open_modal, modal_id: modal_id)
  end

  def upsert_modal(modal_id, partial: nil, locals: {})
    html_content =
ApplicationController.renderer.render(partial: partial, locals:
locals)
    turbo_stream_action_tag(:upsert_modal, modal_id: modal_id,
html: html_content)
  end
end

Turbo::Streams::TagBuilder.prepend(TurboStreamActionsHelper)
```

Include the helper in your controllers.

```
class MyController < ApplicationController

  def my_action
    respond_to do |format|
      format.turbo_stream do
        render turbo_stream: \
          turbo_stream.close_modal
      end
    end
  end
end
```

Using `open_modal` in a controller action:

```
def show
  @item = Item.find(params[:id])
  respond_to do |format|
    format.turbo_stream do
      render turbo_stream: \
        turbo_stream.open_modal("#item_modal")
    end
  end
end
```

Frontend (JavaScript)

Define JavaScript functions to handle the custom Turbo Stream actions.

```
StreamActions.open_modal = function() {
  const modal_id = this.getAttribute("modal_id")
  const modal = document.querySelector(modal_id)
  modal.classList.remove("hidden")
  modal.showModal()
}

StreamActions.upsert_modal = function() {
  const modal_id = this.getAttribute("modal_id")
  const modal_html = this.getAttribute("html")
  const modal = document.querySelector(modal_id)

  if(modal) {
    modal.remove()
  }

  document.body.insertAdjacentHTML("beforeend", modal_html);
}

StreamActions.close_modal = function() {
  document.querySelectorAll("dialog").forEach(dialog => {
    dialog.close()
  })
}
```

Ensure your JavaScript is loaded and accessible. Typically, this code is placed in `app/javascript/packs/application.js` or a similar entry point.

```
import * as Turbo from "@hotwired/turbo"
Turbo.start()

import { StreamActions } from '@hotwired/turbo'

StreamActions.open_modal = function() { ... }
StreamActions.close_modal = function() { ... }
StreamActions.upsert_modal = function() { ... }

window.StreamActions = StreamActions
```

Example HTML for a modal:

```
<dialog id="item_modal" class="hidden">
  <h2>Item Details</h2>
  <p>...</p>
  <button
onclick="document.getElementById('item_modal').close()">Close</butt
on>
</dialog>
```

Using `upsert_modal` to render a partial inside a modal.

```
def new
  @item = Item.new
  respond_to do |format|
    format.turbo_stream do
      render turbo_stream: \
        turbo_stream.upsert_modal("#new_item_modal", partial:
'items/form', locals: { item: @item })
    end
  end
end
```

Make sure your `StreamActions` are globally accessible, or accessible within the scope where Turbo Stream responses are handled. The last line in the first code example `window.StreamActions = StreamActions` is important.

The `turbo_stream_action_tag` method generates the necessary HTML tags.

```
turbo_stream_action_tag(:close_modal)
# => <turbo-stream action="close_modal"></turbo-stream>


turbo_stream_action_tag(:open_modal, modal_id: '#my_modal')
# => <turbo-stream action="open_modal" modal_id="#my_modal">
</turbo-stream>
```

When a `<turbo-stream>` tag with a custom action is processed, Turbo Streams will look for a corresponding function in the `StreamActions` object.

If the function exists, it's executed; otherwise, an error might occur.

Ensure the `ApplicationController.renderer` is configured correctly for rendering partials outside of a normal request/response cycle. Verify that all necessary helpers and context are available within the renderer.

For debugging, use `console.log` within your JavaScript functions to ensure they are being called and that the attributes are being correctly read from the `<turbo-stream>` tags.

When passing data from the backend to the frontend using `turbo_stream_action_tag`, ensure that the data is properly escaped to prevent any potential security vulnerabilities (e.g., XSS attacks).

Use `request.variant = :turbo_stream` in your controller if you want to render different templates based on the request format, for example, render a full page for HTML requests and only the modal content for Turbo Stream requests.

Instead of directly manipulating the DOM, consider using Stimulus controllers within your modals for more complex interactions and state management. This keeps your code organized and maintainable.

When updating a modal with new content, consider using Turbo Frames to update specific parts of the modal instead of replacing the entire modal. This can improve performance and reduce flicker.

Always test your custom Turbo Stream actions thoroughly to ensure they behave as expected in different scenarios. Pay attention to edge cases and potential error conditions.

Consider using custom events to trigger actions within your JavaScript code. This allows for a more decoupled and flexible architecture.

Example of using a custom event to close a modal:

```
document.dispatchEvent(new CustomEvent('close-modal'));



document.addEventListener('close-modal', function() {
  document.querySelectorAll("dialog").forEach(dialog => {
    dialog.close()
  })
});
```

Remember to handle errors gracefully in your JavaScript code. Display user-friendly messages or log errors to the console for debugging purposes.

For more complex modal interactions, consider using a dedicated modal library or component. This can provide additional features and improve the overall user experience.

Ensure that your custom Turbo Stream actions are compatible with different browsers and devices. Test your code on a variety of platforms to ensure a consistent experience.

When using Turbo Streams to update the DOM, be mindful of the potential for race conditions. Ensure that your JavaScript code is properly synchronized to prevent unexpected behavior.

Document your custom Turbo Stream actions thoroughly. This will make it easier for other developers to understand and maintain your code.

Stimulus.js Overview

Introduction to Stimulus

Stimulus is a modest JavaScript framework designed for enhancing HTML with dynamic behavior. It's designed to work well with Turbo, allowing you to build modern web applications without a lot of complex JavaScript.

Key features:

- **Lightweight:** Minimal footprint, easy to learn and use.
- **HTML-centric:** Uses HTML data attributes to bind JavaScript behavior.
- **Complementary to Turbo:** Designed to enhance server-rendered HTML, not replace it.

Stimulus promotes a structured approach to JavaScript in Rails applications, making code more maintainable and easier to understand.

Core Concepts

Controllers	JavaScript objects that manage the behavior of DOM elements. They are the primary building blocks of Stimulus applications.
Targets	Specific DOM elements within a controller's scope that the controller can interact with.
Actions	Methods in the controller that are triggered by DOM events on elements within the controller's scope.
Values	A way to store and manage data within a controller, making it easy to access and update values from the DOM.

Setting Up Stimulus

1. **Install Stimulus:**

```
yarn add @hotwired/stimulus
# or
npm install @hotwired/stimulus
```

2. **Import and Start Stimulus:**

   In your `application.js` or similar entry point:

```
import { Application, Controller }
from "@hotwired/stimulus"

const application =
Application.start()

// Configure Stimulus development
experience
application.debug = false
window.Stimulus   = application

export { application, Controller }
```

3. **Create Controllers Directory:**

   Typically, create a `controllers` directory within your `app/javascript` folder to house your Stimulus controllers.

Simple Stimulus Controller Example

Let's create a simple controller that displays a greeting message.

1. **Create a Controller File:**

   `app/javascript/controllers/hello_controller.js`

```
import { Controller } from
"@hotwired/stimulus"

export default class extends
Controller {
  static targets = [ "name", "output"
]

  greet() {
    this.outputTarget.textContent =
`Hello, ${this.nameTarget.value}!`
  }
}
```

HTML Usage

1. **Add Controller to HTML:**

```
<div data-controller="hello">
  <input data-hello-target="name"
type="text">
  <button data-action="click-
>hello#greet">Greet</button>
  <span data-hello-target="output">
</span>
</div>
```

   Explanation:

   - `data-controller="hello"`: Attaches the `hello` controller to the `div`.
   - `data-hello-target="name"`: Makes the input field accessible as `nameTarget` in the controller.
   - `data-action="click->hello#greet"`: Calls the `greet` method in the `hello` controller when the button is clicked.
   - `data-hello-target="output"`: Makes the span accessible as `outputTarget` in the controller.

Key Data Attributes

`data-controller`
Specifies the Stimulus controller to be associated with the HTML element.
`data-target`
Defines a target element within the controller's scope.
`data-action`
Binds an event to a controller action.
`data-value`
Sets a value on the controller that can be accessed and updated.

Working with Values

Values allow you to store data directly in the DOM and access it from your Stimulus controllers.

Example:

```
//
app/javascript/controllers/counter_contr
oller.js
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  static values = {
    count: { type: Number, default: 0 }
  }

  connect() {
    this.displayCount()
  }

  increment() {
    this.countValue++
    this.displayCount()
  }

  displayCount() {
    this.element.textContent = `Count:
${this.countValue}`
  }
}


<div data-controller="counter" data-
counter-count-value="5">
  <button data-action="click-
>counter#increment">Increment</button>
  <span>Count: 5</span>
</div>
```

Debugging Stimulus

1. **Enable Debug Mode:**

   Set `application.debug = true` in your `application.js` to enable detailed logging.
2. **Use Browser Developer Tools:**

   Inspect the DOM and use `console.log` statements within your controllers to track data and events.
3. **Check for Errors:**

   Pay attention to any error messages in the browser console, as they often indicate issues with your controller logic or HTML bindings.

Stimulus.js Controllers

Controller Basics

Stimulus.js controllers enhance HTML with behavior.

- **Installation:** `yarn add @hotwired/stimulus` or `npm install @hotwired/stimulus`
- **Import:** `import { Controller } from "@hotwired/stimulus"`
- **Create Controller:**

```
//
app/javascript/controllers/example_co
ntroller.js
import { Controller } from
"@hotwired/stimulus"

export default class extends
Controller {
  connect() {
    console.log("Connected to
element!")
  }
}
```

Controllers are defined as ES modules and placed in `app/javascript/controllers` by convention.

Connecting Controllers to HTML

Use `data-controller` attribute to connect a controller to an HTML element.

```
<div data-controller="example">
  <!-- Controller logic applies here -->
</div>
```

- The `data-controller` attribute tells Stimulus to instantiate the `example_controller.js` and associate it with the `<div>`.
- Multiple controllers can be connected to the same element: `data-controller="example another-controller"`

Lifecycle Callbacks

`connect()`

Called when the controller is connected to the DOM.

```
connect() {
  console.log("Controller connected");
}
```

`disconnect()`

Called when the controller is disconnected from the DOM.

```
disconnect() {
  console.log("Controller
disconnected");
}
```

`initialize()`

Called only once when the controller is instantiated.

```
initialize() {
  console.log("Controller initialized");
}
```

Actions

Actions respond to DOM events. Use `data-action` to connect events to controller methods.

```
<button data-action="click-
>example#greet">Greet</button>
```

- `click->example#greet` means on a `click` event, call the `greet` method on the `example` controller.
- Multiple actions can be defined on a single element.

Controller method:

```
greet() {
  alert("Hello!");
}
```

Targets

Targets provide direct references to specific elements within a controller's scope. Define targets in the controller definition, and reference them in your methods.

```
//
app/javascript/controllers/example_contr
oller.js
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  static targets = [ "name", "output" ]

  greet() {
    this.outputTarget.textContent =
`Hello, ${this.nameTarget.value}!`
  }
}
```

HTML:

```
<div data-controller="example">
  <input data-example-target="name"
type="text">
  <button data-action="click-
>example#greet">Greet</button>
  <span data-example-target="output">
</span>
</div>
```

- `data-example-target="name"` makes the input accessible as `this.nameTarget`.

Values

Values allow you to bind data attributes to typed values in your controller.

```
//
app/javascript/controllers/example_contr
oller.js
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  static values = {
    count: Number,
    message: String,
    isActive: Boolean
  }

  connect() {
    console.log(`Initial count:
${this.countValue}`)
  }
}
```

HTML:

```
<div data-controller="example" data-
example-count-value="10" data-example-
message-value="Hello" data-example-is-
active-value="true">
  <!-- Controller logic applies here -->
</div>
```

- Values are automatically converted to the specified type. Available types: `Number`, `String`, `Boolean`, `Array`, `Object`.

CSS Classes

CSS Classes can be toggled on and off using Stimulus. Define class names in the controller, and use the corresponding methods to manipulate them.

```
//
app/javascript/controllers/example_contr
oller.js
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  static classes = [ "hidden" ]

  toggle() {

this.element.classList.toggle(this.hidde
nClass)
  }
}
```

HTML:

```
<div data-controller="example" data-
example-hidden-class="d-none">
  <button data-action="click-
>example#toggle">Toggle</button>
</div>
```

- `data-example-hidden-class="d-none"` sets the `hiddenClass` to `d-none`.

Controller Organization

Organize your Stimulus controllers logically within the `app/javascript/controllers` directory. Consider grouping related controllers into subdirectories.

- **Example:**

```
app/javascript/controllers/
├── form/
│   ├── address_controller.js
│   └── validation_controller.js
├── navigation_controller.js
└── index.js
```

- The `index.js` file is crucial for auto-loading controllers.

```
//
app/javascript/controllers/index.js
import { application } from
"./application"

import ExampleController from
"./example_controller.js"
application.register("example",
ExampleController)
```

Using `this.element`

The `this.element` property in a Stimulus controller refers to the root DOM element to which the controller is attached. It provides a direct way to manipulate the element's attributes, styles, or content.

Example:

```
import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  connect() {
    this.element.classList.add("my-
custom-class")
  }

  disconnect() {
    this.element.classList.remove("my-
custom-class")
  }
}
```

In this example, `this.element` is used to add and remove a CSS class when the controller connects and disconnects, respectively.

Migrating from Rails UJS to Turbo

Why Migrate to Turbo from Rails UJS?

Rails UJS (Unobtrusive JavaScript) is the traditional way Rails handles JavaScript interactions, relying heavily on jQuery. It's being phased out in favor of Hotwire's Turbo for several compelling reasons:

- **Performance:** Turbo significantly reduces the amount of JavaScript needed, leading to faster page loads and a smoother user experience. It updates parts of the page over WebSocket (Turbo Streams) or morphs the DOM directly (Turbo Drive) instead of full page reloads.
- **Modern Approach:** Turbo aligns with modern web development practices by minimizing JavaScript dependencies and leveraging server-side rendering more effectively.
- **Simplicity:** While Rails UJS often involves complex JavaScript setups, Turbo simplifies common interactions with its conventions and minimal configuration.
- **Maintainability:** Less JavaScript means less code to maintain, debug, and test.
- **Hotwire Ecosystem:** Turbo is part of the Hotwire suite, designed to work seamlessly with Stimulus for enhanced front-end interactivity.

Moving to Turbo provides a more streamlined and performant experience by default.

1. **Remove `rails-ujs`:**
   - Remove the `rails-ujs` gem from your `Gemfile`.

```
# Gemfile
# gem 'rails-ujs' # Remove this line
```

   - Remove the corresponding require statement from your `app/assets/javascript/application.js` or `app/javascript/packs/application.js`:

```
// application.js
// require("rails-ujs") // Remove this line
```

   - Run `bundle install` to uninstall the gem.

2. **Install Turbo:**
   - Ensure you have the `turbo-rails` gem in your `Gemfile`:

```
# Gemfile
gem 'turbo-rails'
```

   - Run `bundle install`.
   - Run the Turbo install generator:

```
rails turbo:install
```

   This generator adds Turbo Drive and Turbo Frames to your application.

3. **Replace Data Attributes:**
   - Rails UJS uses data attributes like `data-remote`, `data-method`, `data-confirm`. Turbo uses `data-turbo` attributes.
   - Replace instances of `data-remote="true"` with `data-turbo="true"`.
   - Replace `data-method="[HTTP_METHOD]"` with `data-turbo-method="[HTTP_METHOD]"`.
   - For confirmation dialogs, use `data-turbo-confirm="Are you sure?"` instead of `data-confirm="Are you sure?"`.

4. **Update Form Submissions:**
   - Ensure your forms are submitting correctly with Turbo. By default, Turbo intercepts form submissions and handles them via AJAX.
   - If you need a full page reload for a specific form, add `data-turbo="false"` to the form tag.
   - Use Turbo Streams to update the page in response to form submissions (see Turbo Streams section).

5. **Convert Callbacks:**
   - Rails UJS provides JavaScript callbacks like `ajax:success`, `ajax:error`, and `ajax:complete`. Turbo encourages a different approach using server-sent Turbo Streams or Stimulus controllers.
   - For example, instead of `ajax:success`, use a Turbo Stream to append, prepend, replace, or remove elements on the page after a successful form submission.

6. **Remove jQuery dependency**
   - If your application depends on jQuery, consider refactoring to use vanilla JavaScript with Stimulus. This removes the jQuery dependency.
   - Identify all jQuery usages in your JS code and decide if you can replace them with the default javascript or refactor to Stimulus.

Code Examples: Before and After

Rails UJS (Before)

```
<%= link_to 'Delete', item_path(item), method:
:delete, data: { confirm: 'Are you sure?' } %>
```

Turbo (After)

```
<%= link_to 'Delete', item_path(item), data: { turbo_method: :delete,
turbo_confirm: 'Are you sure?' } %>
```

Rails UJS (Before)

```
<%= form_with(model: @article, remote: true) do |form|
%>
  ...
<% end %>
```

Turbo (After)

```
<%= form_with(model: @article) do |form| %>
  ...
<% end %>
```

Note: Turbo handles the form submission via AJAX by default. Use Turbo Streams in your controller to update the view after submission.

Rails UJS (Before - Custom JS Callback)

```
$('form').on('ajax:success', function(event) {
  // Handle success
});
```

Turbo (After - Turbo Stream)

Controller (example):

```
# app/controllers/articles_controller.rb
def create
  @article = Article.new(article_params)
  if @article.save
    render turbo_stream: turbo_stream.append('articles', partial: 'article',
locals: { article: @article })
  else
    render :new
  end
end
```

View (articles/create.turbo_stream.erb):

```
<%= turbo_stream.append "articles" do %>
  <%= render @article %>
<% end %>
```

Rails UJS (Before - data-disable-with)

```
<%= button_tag 'Submit', data: { disable_with:
'Processing...' } %>
```

Turbo (After - data-turbo-submits-with)

```
<%= button_tag 'Submit', data: { turbo_submits_with: 'Processing...' } %>
```

Rails UJS (Before - remote link with GET method)

```
<%= link_to 'Get Data', data_path, remote: true,
method: :get %>
```

**Turbo (After - using data-turbo)

```
<%= link_to 'Get Data', data_path, data: {turbo: true, turbo_method: :get}
%>
```

Rails UJS (Before - prevent full page reload)

```
$('a[data-remote]').on('ajax:beforeSend', function() {
  // Custom logic before sending the request
});
```

Turbo (After - using Turbo events and `preventDefault()` in Stimulus)

```
// app/javascript/controllers/link_controller.js
import { Controller } from "@hotwired/stimulus"

export default class extends Controller {
  connect() {
    this.element.addEventListener("turbo:before-fetch-request",
this.beforeFetchRequest.bind(this))
  }

  beforeFetchRequest(event) {
      //Custom logic before sending the request
  }
}
```

Action Cable Integration for Turbo Streams

Overview

Integrating Action Cable with Turbo Streams allows you to broadcast real-time updates to your Rails application. This enables features like live comments, real-time notifications, and dynamic content updates without full page reloads. Turbo Streams handle the rendering and updating of specific DOM elements, while Action Cable provides the real-time communication infrastructure.

This integration involves configuring Action Cable, setting up Turbo Streams, and broadcasting updates from your Rails backend to connected clients.

Configuration

1. Configure Action Cable:	Ensure Action Cable is properly configured in your `config/cable.yml` file. Typically, you'll use Redis for production environments.
2. Setup Redis (optional):	If using Redis, ensure it's running and accessible to your Rails application. Update `config/cable.yml` to point to your Redis instance.
3. Mount Action Cable:	Mount Action Cable in your `config/routes.rb` file: `mount ActionCable.server => '/cable'`

Creating a Channel

Generate a channel using Rails generators. For example, to create a `CommentsChannel`:

```
rails generate channel Comments
```

This creates `app/channels/comments_channel.rb` and related files. Modify the channel to handle subscriptions:

```
class CommentsChannel <
ApplicationCable::Channel
  def subscribed
    stream_from "comments"
  end

  def unsubscribed
    # Any cleanup logic when user
unsubscribes
  end
end
```

Broadcasting Turbo Streams

To broadcast Turbo Streams, use the `turbo_stream` methods within your Rails controllers or models.

Example: Broadcasting a new comment:

```
class CommentsController <
ApplicationController
  def create
    @comment =
Comment.new(comment_params)
    if @comment.save

Turbo::StreamsChannel.broadcast_replace_
later_to(
        "comments",
        target:   "new_comment",
        partial:  "comments/form",
        locals: {comment: Comment.new}
      )

Turbo::StreamsChannel.broadcast_prepend_
later_to(
        "comments",
        target:   "comments",
        partial:  "comments/comment",
        locals: {comment: @comment}
      )
      head :ok
    else
      render :new, status:
:unprocessable_entity
    end
  end

  private

  def comment_params

params.require(:comment).permit(:content
)
  end
end
```

In this example, `turbo_stream.prepend` adds a new comment to the `#comments` div, and the `turbo_stream.replace` replaces the form with empty one. Make sure you create partials:

`app/views/comments/_comment.html.erb`

```
<%= turbo_frame_tag dom_id(comment) do
%>
  <p><%= comment.content %></p>
<% end %>
```

`app/views/comments/_form.html.erb`

```
<%= turbo_frame_tag "new_comment" do %>
  <%= form_with(model: comment, url:
comments_path) do |form| %>
    <%= form.text_area :content %>
    <%= form.submit "Add Comment" %>
  <% end %>
<% end %>
```

Client-Side Subscription

On the client-side, subscribe to the Action Cable channel using JavaScript. This is typically done in your `app/javascript/channels/comments_channel.js` file:

```
import consumer from "channels/consumer"

consumer.subscriptions.create("CommentsC
hannel", {
  connected() {
    console.log("Connected to the
comments channel");
    // Called when the subscription is
ready for use on the server
  },

  disconnected() {
    console.log("Disconnected from the
comments channel");
    // Called when the subscription has
been terminated by the server
  },

  received(data) {
    console.log("Received data: ",
data);
    // Called when there's incoming data
on the websocket for this channel
  }
});
```

Ensure that this JavaScript file is included in your application's JavaScript bundle.

Displaying Turbo Streams

Make sure that the container to update exists. The container is `turbo_frame_tag`:

```
<div id="comments">
  <%= render @comments %>
</div>
<%= render "comments/form", comment:
Comment.new %>
```

When the broadcast reaches the client, Turbo Streams automatically updates the DOM based on the specified actions (e.g., `prepend`, `append`, `replace`).

Example: Real-time Notifications

You can adapt the same principles to implement real-time notifications. Create a `NotificationsChannel`, broadcast updates when new notifications are created, and update the notification list on the client-side.

```
Turbo::StreamsChannel.broadcast_prepend_
later_to(
  "notifications",
  target: "notifications",
  partial: "notifications/notification",
  locals: { notification: @notification
}
)
```

Ensure your client-side JavaScript subscribes to the `NotificationsChannel` to receive these updates.

Debugging Techniques and Error Handling

General Debugging Strategies

When debugging Hotwire applications, it's crucial to use browser developer tools and server logs effectively.

- **Inspect Network Requests:** Check the network tab for Turbo Drive requests and Turbo Stream responses.
- **Use `console.log`:** Add `console.log` statements in your Stimulus controllers to track variable values and execution flow.
- **Examine Server Logs:** Monitor Rails server logs for any errors or unexpected behavior.
- **Rails Debugger:** Utilize `debugger`, `byebug` or `pry` for step-by-step debugging on the server side.

Set the `data-turbo-action` attribute to `advance`, `replace`, or `restore` on links and forms to explicitly control navigation behavior. This can help identify unexpected navigation actions.

Debugging Turbo Frames

Turbo Frames can sometimes behave unexpectedly if the frame IDs are not unique or if the server response doesn't match the expected frame. Here's how to debug them:

- **Check Frame IDs:** Ensure that all Turbo Frame IDs on a page are unique to avoid conflicts.
- **Verify Server Response:** Use browser developer tools to inspect the HTML returned in the Turbo Frame response. Make sure it contains the correct content wrapped in the corresponding `<turbo-frame>` tag.
- **Inspect `turbo:frame-load` event:** Use event listeners for debugging.
- **Check for nested frames:** Ensure proper nesting of the frames.

Example of using `turbo:frame-load` event listener:

```
document.addEventListener('turbo:frame-load', (event) => {
  console.log('Turbo Frame loaded:', event.target.id);
});
```

Sometimes, content might not load into a Turbo Frame due to JavaScript errors. Check the browser console for JavaScript errors that might be preventing the frame from loading correctly.

Debugging Turbo Streams

Turbo Streams are used to update parts of the page dynamically. Debugging them involves checking the server response and ensuring the correct stream actions are being applied.

- **Inspect Stream Response:** Check the content type of the response. It should be `text/vnd.turbo-stream.html`.
- **Verify Stream Actions:** Ensure the Turbo Stream response contains valid `<turbo-stream>` elements with the correct action (e.g., `append`, `prepend`, `replace`, `remove`) and target.
- **Use `turbo:before-stream-render` event:** Add event listeners for debugging.
- **Check for correct target:** The target attribute in the stream must match existing element id on the page

Example of using `turbo:before-stream-render` event listener:

```
document.addEventListener('turbo:before-stream-render', (event)
=> {
  console.log('Turbo Stream action:', event.detail.action);
  console.log('Turbo Stream target:', event.detail.target);
});
```

If a Turbo Stream action is not being applied, check the browser console for JavaScript errors. Also, ensure that the target element exists on the page and that the stream action is valid for that element.

Error Handling in Turbo Streams

Implement error handling on the server-side to gracefully handle exceptions and return appropriate Turbo Stream responses.

- **Rescue Exceptions:** Use `rescue_from` in your Rails controllers to catch exceptions and render error messages as Turbo Streams.
- **Return Error Streams:** Return Turbo Stream responses that display error messages in the UI.
- **Log Errors:** Log detailed error messages on the server for debugging purposes.

Example of using `rescue_from` in Rails controller:

```
class CommentsController < ApplicationController
  rescue_from StandardError, with: :render_error

  def create
    @comment = Comment.new(comment_params)
    if @comment.save
      render turbo_stream: ... # Stream for successful comment
creation
    else
      render turbo_stream:
render_error(@comment.errors.full_messages.join(', '))
    end
  end

  private

  def render_error(message)
    turbo_stream.replace("error_messages", partial:
"shared/error_messages", locals: { errors: message })
  end
end
```

In some cases, you may want to prevent Turbo Drive from navigating to a new page on error. You can do this by canceling the `turbo:before-visit` event.

```
document.addEventListener('turbo:before-visit', (event) => {
  if (someConditionThatIndicatesAnError()) {
    event.preventDefault();
  }
});
```

Stimulus Controller Debugging

Stimulus controllers manage the behavior of your HTML elements. Debugging them involves ensuring that the controller is correctly connected and that actions are being triggered as expected.

- **Check Controller Connection:** Verify that the Stimulus controller is correctly connected to the HTML element using the `data-controller` attribute.
- **Inspect Action Bindings:** Ensure that actions are correctly bound to the controller methods using the `data-action` attribute.
- **Use `console.log` :** Add `console.log` statements in your controller methods to track the execution flow and variable values.
- **Browser Debugger:** Use the browser's debugger to step through the controller code and inspect the state of the application.

```
<div data-controller="my-controller">
  <button data-action="click->my-controller#handleClick">Click
Me</button>
</div>


// my_controller.js
import { Controller } from "@hotwired/stimulus"

export default class extends Controller {
  connect() {
    console.log("MyController connected");
  }

  handleClick() {
    console.log("Button clicked");
  }
}
```

If a Stimulus action is not being triggered, check the spelling of the controller and method names in the `data-action` attribute. Also, ensure that the controller is correctly connected to the HTML element.

Integration Testing

Write integration tests to ensure that your Hotwire components are working correctly together. Use Capybara or Rails system tests to simulate user interactions and verify the behavior of your application.

- **Simulate User Interactions:** Use Capybara methods to simulate user interactions, such as clicking buttons and filling in forms.
- **Assert on Page Content:** Use Capybara assertions to verify that the page content is being updated correctly by Turbo Streams and Turbo Frames.
- **Check for JavaScript Errors:** Use `page.driver.browser.logs` to check for JavaScript errors in your tests.

Example of a Rails system test:

```
require "application_system_test_case"

class CommentsTest < ApplicationSystemTestCase
  test "creating a comment"
    visit post_path(posts(:one))
    fill_in "comment_body", with: "This is a test comment"
    click_on "Create Comment"

    assert_text "This is a test comment"
  end
end
```

When writing integration tests, make sure to wait for Turbo Streams and Turbo Frames to load before asserting on the page content. Use Capybara's `assert_selector` with a `wait` option to wait for elements to appear on the page.

Turbo Caching and Fallback Behaviors

Understanding Turbo Caching

Turbo Drive automatically caches pages as you visit them, making subsequent visits feel instantaneous. It leverages the browser's history API to achieve this.

When you navigate back or forward, Turbo Drive restores the page from its cache, avoiding a full page reload.

This caching mechanism primarily targets GET requests. POST, PUT, and DELETE requests trigger a full page reload to ensure data consistency.

You can control Turbo's caching behavior using meta tags in your HTML. For instance, to disable caching for a specific page:

```
<meta name="turbo-cache-control" content="no-cache">
```

This tag instructs Turbo not to cache the current page, ensuring it's always fetched from the server.

Alternatively, you can use `turbo-cache-control: public` to explicitly allow caching, which is useful if you have a global `no-cache` setting.

Turbo also respects standard HTTP caching headers like `Cache-Control` and `Expires`.

For dynamic content, consider using `Vary` header to cache different versions of a page based on user-specific information (e.g., user ID, authentication status).

If you are using `Turbolinks` gem, please consider removing it. `Turbo` already covers all the features of `Turbolinks`.

Debugging Turbo Caching Issues

When encountering unexpected caching behavior, use your browser's developer tools to inspect the HTTP cache and verify that pages are being cached and served correctly.

Check the `turbo-cache-control` meta tags and HTTP caching headers to ensure they are configured as intended.

Clear your browser's cache and cookies to rule out any stale or corrupted cached data.

Use the `Turbo.clearCache()` method to programmatically clear Turbo's cache during development or testing.

Inspect the network requests in the developer tools to see if pages are being fetched from the server or served from the cache.

Verify that your server is sending the correct `Content-Type` headers for your responses, as incorrect headers can interfere with caching.

If you're using a CDN, check its caching configuration and ensure it's properly caching your assets.

Consider using a logging framework to track Turbo-related events and identify potential caching issues.

Handling Cache Fallbacks

In scenarios where the Turbo cache is unavailable (e.g., due to a network error), you can implement fallback behaviors to ensure a smooth user experience.

One approach is to detect Turbo availability and conditionally perform a full page reload if Turbo is not present:

```
if (window.Turbo == null) {
  window.location.reload()
}
```

This JavaScript snippet checks if the `Turbo` object is defined. If not, it triggers a standard page reload, bypassing Turbo's caching mechanism.

Another strategy involves using service workers to cache critical assets and provide offline access. Service workers can intercept network requests and serve cached content when the network is unavailable.

When implementing service workers, ensure they are compatible with Turbo Drive's caching strategy to avoid conflicts or unexpected behavior.

Consider using a combination of Turbo Drive caching and service worker caching for optimal performance and resilience.

If you are using CDN - ensure that proper headers are configured, to enable caching mechanism on the CDN level.

Fine-Grained Control with Turbo Streams

Turbo Streams offer a more granular way to update specific parts of a page without requiring a full reload. This can be particularly useful for dynamic content updates.

You can use Turbo Streams to broadcast changes to connected clients via WebSocket connections, ensuring real-time updates.

To use Turbo Streams, you typically define actions in your Rails controllers that render Turbo Stream templates. For example:

```
# app/controllers/comments_controller.rb
def create
  @comment = Comment.new(comment_params)
  if @comment.save
    render turbo_stream: turbo_stream.append("comments",
partial: "comments/comment", locals: { comment: @comment })
  else
    render :new, status: :unprocessable_entity
  end
end
```

This code appends a new comment to the `#comments` element on the page when a comment is successfully created.

Ensure that your Turbo Stream actions are idempotent to handle potential duplicate broadcasts or re-renders.

Consider using `turbo_stream.replace` or `turbo_stream.update` to modify existing elements on the page, providing a seamless user experience.

Always test Turbo Streams with different network conditions to ensure they function correctly under varying latency and bandwidth scenarios.

Turbo Frames for Modular Content

Turbo Frames allow you to isolate sections of a page into independent, cacheable units. This is useful for creating modular and reusable components.

Each Turbo Frame has a unique `id` attribute, which Turbo uses to identify and update the frame's content.

When a link or form within a Turbo Frame is clicked or submitted, Turbo Drive only updates the content of that specific frame, leaving the rest of the page untouched.

Here's an example of a Turbo Frame:

```
<turbo-frame id="user_profile">
  <!-- User profile content here -->
</turbo-frame>
```

If the content inside a `<turbo-frame>` tag is not immediately available, you can use the `loading` attribute to specify a placeholder or loading indicator.

```
<turbo-frame id="comments" loading="lazy">
  <template>
    Loading comments...
  </template>
</turbo-frame>
```

Use Turbo Frames to break down complex pages into smaller, manageable components, improving performance and maintainability.

Nested Turbo Frames are supported, allowing for even greater flexibility in structuring your application's UI.

Hotwire Security Considerations

CSRF Protection in Hotwire

Rails' built-in CSRF protection works seamlessly with Hotwire. Ensure `protect_from_forgery` is included in your `ApplicationController`.

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
end
```

When using Turbo Streams with forms, Rails automatically includes the CSRF token in the form data. No extra steps are required for standard form submissions.

For non-standard form submissions (e.g., AJAX-like requests with Turbo Streams), ensure the CSRF token is included in the request headers. You can do this via JavaScript:

```
// Example using Fetch API
fetch('/your_endpoint', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'X-CSRF-Token': document.querySelector('meta[name="csrf-token"]').content
  },
  body: JSON.stringify({ key: 'value' })
});
```

Secure Handling of Turbo Streams

Always validate and sanitize data received via Turbo Streams to prevent injection attacks. Ensure that only authorized users can modify specific parts of the page.

Avoid directly embedding user-supplied data into Turbo Stream actions without proper escaping. Use Rails' built-in sanitization methods.

When rendering Turbo Streams in the controller, use the `escape: false` option with caution. Only use it if you are absolutely sure the content is safe.

```
# Example: Rendering a Turbo Stream (be cautious with user input)
render turbo_stream: turbo_stream.replace(
  'element_id',
  partial: 'your_partial', # Make sure this partial sanitizes data
  locals: { data: @safe_data } # Only pass safe data here
)
```

Implement proper authentication and authorization checks in your controllers to ensure that users can only access or modify resources they are permitted to.

Use strong parameters to whitelist attributes that can be updated via form submissions or API requests. This helps prevent mass assignment vulnerabilities.

Content Security Policy (CSP)

Configure a Content Security Policy (CSP) to mitigate the risk of Cross-Site Scripting (XSS) attacks. CSP allows you to define which sources of content (scripts, stylesheets, images, etc.) the browser should trust.

In your `ApplicationController`, set the CSP header:

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  before_action :set_content_security_policy

  def set_content_security_policy
    response.headers['Content-Security-Policy'] = "script-src 'self' https://cdn.example.com; object-src 'none';"
  end
end
```

Adjust the CSP directives based on your application's needs. Common directives include `script-src`, `style-src`, `img-src`, and `default-src`. Use `'self'` to allow content from the same origin.

Be mindful of inline scripts and styles, as they are often blocked by CSP unless you use `'unsafe-inline'` (which reduces security). Consider using nonces or hashes for inline scripts if necessary.

When using Stimulus, ensure that your CSP allows the necessary JavaScript files to be loaded and executed. You may need to whitelist the CDN or domain where your Stimulus code is hosted.

Regularly review and update your CSP to address new threats and ensure compatibility with your application's dependencies.

Hotwire Performance Tips & Pitfalls

Optimize Turbo Streams

Partial Page Updates:
Only update the necessary parts of the page with Turbo Streams to minimize data transfer and rendering time.

```
# Avoid this:
render turbo_stream: turbo_stream.replace('content', partial:
'full_content')

# Prefer this:
render turbo_stream: turbo_stream.replace('partial_area',
partial: 'specific_partial')
```

Batch Turbo Streams:
Combine multiple Turbo Stream actions into a single response to reduce latency.

```
render turbo_stream: [turbo_stream.append('list', partial:
'item1'),
                      turbo_stream.append('list', partial:
'item2')]
```

Use `turbo_frame_tag` Wisely:
Ensure frames are appropriately scoped to avoid unnecessary re-rendering.

```
<%= turbo_frame_tag 'comments' do %>
  <%= render @comments %>
<% end %>
```

Lazy Loading:
Defer loading of non-critical content within Turbo Frames until they are visible.

```
<turbo-frame id="lazy_content" loading="lazy"
src="/lazy_content_path"></turbo-frame>
```

Avoid Excessive DOM Manipulation:
Limit the number of DOM changes triggered by Turbo Streams to reduce browser overhead. Profile with browser dev tools.

Common Pitfalls

Over-reliance on Turbo Frames:
Using too many Turbo Frames can lead to increased complexity and potential performance issues. Profile each frame's impact.

Ignoring Network Latency:
Consider network latency when designing interactions. Optimize for fewer round trips.

Unoptimized Images:
Ensure images are optimized for the web to reduce page load time. Use tools like ImageOptim or services like Cloudinary.

Excessive JavaScript:
Minimize the amount of JavaScript code to reduce parsing and execution time. Use code splitting and lazy loading.

Neglecting Browser Caching:
Leverage browser caching to reduce the number of requests to the server. Configure appropriate cache headers.

Stimulus Controller Optimization

Debounce Actions:
Use `debounce` to prevent rapid firing of actions, especially on user input.

```
// Example:
// app/javascript/controllers/search_controller.js
import { Controller } from "@hotwired/stimulus"
import { debounce } from "debounce";

export default class extends Controller {
  static targets = [ "input", "results" ]

  initialize() {
    this.search = debounce(this.search, 300).bind(this)
  }

  search() {
    // Perform search logic here
  }
}
```

Disconnect Observers:
Disconnect MutationObservers when the controller is disconnected to prevent memory leaks.

```
// Example
disconnect() {
  if (this.observer) {
    this.observer.disconnect()
  }
}
```

Efficient Data Attributes:
Use data attributes efficiently to pass data to Stimulus controllers, avoiding unnecessary DOM reads.

```
<div data-controller="example" data-example-url-
value="/api/data"></div>
```

Lazy Initialization:
Initialize Stimulus controllers only when they are needed to reduce initial page load time.

Reduce DOM Queries:
Cache frequently accessed DOM elements to minimize the number of DOM queries.

```
connect() {
  this.cachedElement = this.element.querySelector('.my-
element');
}
```

Server-Side Performance

Optimize Database Queries:
Ensure database queries are efficient to reduce server response time.

```
# Avoid N+1 queries:
@posts = Post.all # Causes N+1 issue when accessing user for each post

# Prefer eager loading:
@posts = Post.includes(:user).all # Solves N+1 issue
```

Caching:
Implement caching strategies (e.g., fragment caching, Rails.cache) to reduce database load.

```
# Fragment caching
<% cache @post do %>
  <%= render @post %>
<% end %>
```

Background Jobs:
Offload time-consuming tasks to background jobs to improve responsiveness.

```
# Example using ActiveJob
class ProcessDataJob < ApplicationJob
  queue_as :default

  def perform(data)
    # Process data here
  end
end

ProcessDataJob.perform_later(data)
```

Efficient Rendering:
Optimize view rendering by avoiding complex logic in views and using partials effectively.

Use Indexes:
Ensure proper database indexes are in place for frequently queried columns.

Benchmarking and Profiling

Rails Benchmark Tool:
Use the Rails benchmark tool to measure the performance of different code paths.

```
require 'benchmark'

n = 5000
Benchmark.bm do |x|
  x.report('each:')   { n.times do   a = []; (1..1000).each {|i| a << i} end }
  x.report('times:')  { n.times do a = []; 1000.times {|i| a << i} end }
end
```

Browser Developer Tools:
Use browser developer tools (e.g., Chrome DevTools) to profile JavaScript execution, network requests, and rendering performance.

Bullet Gem:
Use the Bullet gem to detect N+1 queries and other performance issues.

```
# Add to Gemfile
gem 'bullet'
```

Rack Mini Profiler:
Use Rack Mini Profiler to profile Rack middleware and database queries.

```
# Add to Gemfile
gem 'rack-mini-profiler'
```

Memory Profiling:
Profile memory usage to identify memory leaks and optimize memory allocation.

Turbo Native Considerations

Optimize Asset Size:
Reduce the size of assets (CSS, JavaScript, images) to minimize download time on mobile devices.

Native Bridge Overhead:
Be mindful of the overhead of communication between the native app and the web view. Minimize the number of bridge calls.

Offline Support:
Implement offline support to improve the user experience in areas with poor connectivity. Use service workers to cache assets and data.

Adaptive Content:
Serve different content based on the device's capabilities and network conditions.

Preloading:
Preload critical assets to reduce perceived latency.

Hotwire Common Patterns & Best Practices

Turbo Drive Best Practices

Use Turbo Drive for standard navigation:
Turbo Drive automatically intercepts link clicks and form submissions, turning them into AJAX requests. This provides a faster, more responsive user experience by updating only the changed parts of the page.

```
<%= link_to 'Home', root_path %>
```

Ensure idempotent GET requests:
GET requests should not have side effects. This ensures that Turbo Drive's caching and preloading mechanisms work correctly without unintended consequences.

```
# Avoid operations that modify data in GET requests
def show
  @item = Item.find(params[:id])
end
```

Use `turbo_frame_tag` for isolated updates:
Wrap sections of your page in `turbo_frame_tag` to target specific areas for updates after form submissions or other actions.

```
<%= turbo_frame_tag 'comments' do %>
  <%= render @comments %>
<% end %>
```

Handle Turbo Drive events:
Listen for Turbo Drive events like `turbo:before-fetch-request`, `turbo:before-render`, and `turbo:load` to perform custom actions during the page lifecycle.

```
document.addEventListener('turbo:load', () => {
  console.log('Page loaded via Turbo Drive');
});
```

Leverage Turbo Streams for real-time updates:
Use Turbo Streams to broadcast changes to the DOM from the server, enabling real-time updates without full page reloads.

```
# Broadcast a new comment
after_create_commit do
  broadcast_append_to 'comments', target: 'comments', partial: 'comments/comment'
end
```

Turbo Frames Patterns

Nested Frames:
Nest `turbo_frame_tag` elements to create independent, updatable regions within a page. This allows for granular control over updates.

```
<%= turbo_frame_tag 'outer_frame' do %>
  <%= turbo_frame_tag 'inner_frame' do %>
    Content for inner frame
  <% end %>
<% end %>
```

Lazy Loading with Frames:
Use `turbo_frame_tag` with the `loading: :lazy` attribute to load content only when the frame is scrolled into view, improving initial page load times.

```
<%= turbo_frame_tag 'lazy_frame', loading: :lazy, src: lazy_content_path %>
```

Targeting Frames from Forms:
Specify the `turbo_frame` option in form helpers to target a specific frame for updates upon form submission.

```
<%= form_with(model: @post, turbo_frame: 'post_form') do |form| %>
  ...
<% end %>
```

Using `turbo-frame` Attributes:
Utilize attributes like `turbo-frame` on links and forms to specify the target frame directly in HTML.

```
<a href="/items/1" turbo-frame="item_details">Show Item</a>
```

Fallback Content:
Provide fallback content within a `turbo_frame_tag` that is displayed if Turbo Drive is not available or if the frame fails to load.

```
<%= turbo_frame_tag 'my_frame' do %>
  <p>Loading...</p>
  <template data-turbo-frame="my_frame">
    <p>Content loaded!</p>
  </template>
<% end %>
```

Turbo Streams Strategies

Broadcast Updates from Models:
Use `after_create_commit`, `after_update_commit`, and `after_destroy_commit` callbacks in your models to automatically broadcast updates via Turbo Streams.

```
class Comment < ApplicationRecord
  after_create_commit { broadcast_append_to 'comments' }
  after_update_commit { broadcast_replace_to 'comments' }
  after_destroy_commit { broadcast_remove_to 'comments' }
end
```

Target Specific Users or Groups:
Customize the Turbo Stream channel to target specific users or groups, enabling private or segmented real-time updates.

```
broadcast_append_to "user_#{user.id}_notifications", target: 'notifications', partial: 'notifications/notification'
```

Use `turbo_stream_from` in Views:
Subscribe to a Turbo Streams channel in your views using `turbo_stream_from` to receive updates broadcast to that channel.

```
<%= turbo_stream_from 'comments' %>
```

Custom Turbo Stream Actions:
Define custom Turbo Stream actions to perform specific DOM manipulations beyond the built-in actions (append, prepend, replace, remove, etc.).

```
// Example: Custom action to highlight an element
Turbo.StreamActions.highlight = function() {

this.target.classList.add('highlighted')
;
};
```

Conditional Broadcasting:
Broadcast Turbo Streams conditionally based on certain criteria, such as user roles or application state.

```
# Broadcast only if the comment is approved
after_create_commit do
  broadcast_append_to 'comments' if approved?
end
```

Stimulus Controller Conventions

Naming Conventions:
Use descriptive names for your Stimulus controllers that reflect their purpose. Follow the `[namespace]--[component]--controller` naming convention.

```
// app/javascript/controllers/search--filter_controller.js
import { Controller } from "@hotwired/stimulus"

export default class extends Controller {
  connect() {
    console.log("Connected search filter controller");
  }
}
```

Data Attributes for Targets and Actions:
Use `data-controller`, `data-target`, and `data-action` attributes to connect your HTML elements to your Stimulus controllers.

```
<div data-controller="search--filter">
  <input type="text" data-search--filter-target="input">
  <button data-action="search--filter#search">Search</button>
</div>
```

Controller Organization:
Organize your Stimulus controllers into logical directories based on their functionality or the components they control.

```
app/javascript/controllers/
├── search/
│   ├── filter_controller.js
│   └── autocomplete_controller.js
└── form/
    └── validation_controller.js
```

Lifecycle Callbacks:
Leverage Stimulus lifecycle callbacks like `connect()`, `disconnect()`, and `initialize()` to manage the state and behavior of your controllers.

```
export default class extends Controller {
  connect() {
    // Called when the controller is connected to the DOM
  }

  disconnect() {
    // Called when the controller is disconnected from the DOM
  }
}
```

Debouncing and Throttling:
Use debouncing or throttling techniques to optimize the performance of event handlers, especially for frequently triggered events like `input` or `scroll`.

```
import { debounce } from 'debounce';

export default class extends Controller {
  connect() {
    this.search = debounce(this.search.bind(this), 300);
  }

  search() {
    // Perform search logic
  }
}
```

Stimulus Advanced Patterns

Using Stores for Shared State:
Implement stores (using libraries like `nanostores`) to manage and share state between multiple Stimulus controllers.

```
// store.js
import { atom } from 'nanostores'

export const count = atom(0)

// Controller A
import { count } from '../store'

export default class extends Controller {
  increment() {
    count.set(count.get() + 1)
  }
}
```

Asynchronous Actions:
Handle asynchronous operations within your Stimulus actions using `async/await` or Promises.

```
export default class extends Controller {
  async loadData() {
    const response = await fetch('/data');
    const data = await response.json();
    // Update the DOM with the data
  }
}
```

Dynamic Targets:
Dynamically add or remove targets based on application state or user interactions.

```
export default class extends Controller {
  addTarget() {
    const newElement = document.createElement('div');
    newElement.dataset.target = 'my-controller.dynamicTarget';

    this.element.appendChild(newElement);
    this.targets.refresh(); // Refresh the targets cache
  }
}
```

External Libraries Integration:
Integrate external JavaScript libraries and frameworks with Stimulus controllers to enhance functionality.

```
import Chart from 'chart.js';

export default class extends Controller
{
  connect() {
    new Chart(this.element, {
      type: 'bar',
      data: { ... }
    });
  }
}
```

Form Submission Handling with Stimulus:
Use Stimulus to handle form submissions, including validation and AJAX requests.

```
<form data-controller="form-submit"
data-action="submit->form-
submit#submit">
  <!-- Form fields -->
  <button type="submit">Submit</button>
</form>


import { Controller } from
"@hotwired/stimulus"

export default class extends Controller
{
  submit(event) {
    event.preventDefault()
    fetch(this.element.action, {
      method: this.element.method,
      body: new FormData(this.element)
    })
    .then(response => response.json())
    .then(data => {
      // Handle the response data
    })
  }
}
```

Performance Optimization

Minimize DOM Manipulations:
Reduce the number of DOM manipulations by batching updates and using techniques like `requestAnimationFrame`.

```
requestAnimationFrame(() => {
  // Perform multiple DOM updates here
});
```

Optimize Images and Assets:
Optimize images and other assets to reduce page size and improve load times.

- Use appropriate image formats (e.g., WebP).
- Compress images without losing quality.
- Use a CDN to serve assets from geographically distributed servers.

Lazy Loading:
Implement lazy loading for images and other content that is not immediately visible on the page.

```
<img src="placeholder.png" data-
src="actual-image.jpg" loading="lazy">
```

Caching Strategies:
Implement caching strategies to reduce server load and improve response times.

- Use HTTP caching headers to cache static assets in the browser.
- Implement server-side caching using Rails' caching mechanisms.

Code Splitting:
Split your JavaScript code into smaller chunks to reduce the initial load time.

- Use tools like Webpack or Rollup to split your code into separate bundles.
- Load only the necessary code for each page or component.

Testing Hotwire Applications

System Tests with Capybara:
Use system tests with Capybara to test the full integration of Hotwire components, ensuring that Turbo Drive, Turbo Frames, and Stimulus controllers work together correctly.

```
require "application_system_test_case"

class ItemsTest <
ApplicationSystemTestCase
  test "visiting the index" do
    visit items_url

    assert_selector "h1", text: "Items"
  end
end
```

JavaScript Testing with Jest or Mocha:
Test your Stimulus controllers using JavaScript testing frameworks like Jest or Mocha to ensure that they behave as expected.

```
// Example Jest test
import { Application } from
"@hotwired/stimulus"
import HelloController from
"../../javascript/controllers/hello_cont
roller"

describe("HelloController", () => {
  it("should display a greeting", () =>
{
    document.body.innerHTML = '<div
data-controller="hello" data-hello-
target="name"></div>'
    const application =
Application.start()
    application.register("hello",
HelloController)
    const element =
document.querySelector("[data-hello-
target='name']")

expect(element.textContent).toEqual("")
  })
})
```

Integration Tests for Turbo Streams:
Write integration tests to verify that Turbo Streams are being broadcast and received correctly, and that the DOM is being updated as expected.

```
# Example RSpec integration test
it "broadcasts a new comment" do
  expect {
    post comments_path, params: {
comment: { content: "New comment" } }
  }.to change { Comment.count }.by(1)

  assert_turbo_stream action: :append,
target: "comments"
end
```

Mocking Turbo Streams in Tests:
Mock Turbo Streams in tests to isolate the components being tested and avoid dependencies on external systems.

```
# Example RSpec mock
allow(Turbo::StreamsChannel).to
receive(:broadcast_append_to)
```

Using `assert_select_turbo_stream` Helper:
Utilize the `assert_select_turbo_stream` helper in Rails system tests to assert that Turbo Stream actions are being performed correctly.

```
assert_select_turbo_stream action:
:replace, target: "comment_1" do
  assert_select "p", text: "Updated
comment content"
end
```

Hotwire custom actions and integration with View Components

Turbo Messages

```
    def update_unread_messages_count(user)
      messages_count = user.messages.unread.count

      turbo_stream_sidebar_actions = [
        Turbo::StreamsChannel.turbo_stream_action_tag(
          :update_unread_messages_count,
          value: messages_count
        ),
      ]
      Turbo::StreamsChannel.broadcast_stream_to([user,
:notifications], content:
turbo_stream_sidebar_actions.join("\n"))
    end
```

You can pass many custom actions to `turbo_stream_sidebar_actions` and they will be broadcasted in a single message.

Nex this stream action will be captured on JS side

```
StreamActions.update_unread_messages_count = function () {
  const unreadMessagesCount = document.getElementById("unread-
messages-count");
  const messagesCount = parseInt(this.getAttribute("value")) ||
0;

  if (notificationsCount > 0) {
    unreadMessagesCount.textContent = messagesCount;
    unreadMessagesCount.classList.remove("hidden");
  } else {
    unreadMessagesCount.textContent = "";
    unreadMessagesCount.classList.add("hidden");
  }
}
```

With View Components

```
message_component = Chat::MessageComponent.new(
      viewer: user,
      message: self
    )

    turbo_stream_sidebar_actions = [
      Turbo::StreamsChannel.turbo_stream_action_tag(
        :remove,
        target: "sidebar-message-#{message.id}",
      ),
      Turbo::StreamsChannel.turbo_stream_action_tag(
        :prepend,
        target: "sidebar-messages",
        template:
ApplicationController.render(message_component, layout: false),
      ),

Turbo::StreamsChannel.turbo_stream_action_tag(:highlight_message
),
    ]

    Turbo::StreamsChannel.broadcast_stream_to([:sidebar, user],
content: turbo_stream_sidebar_actions.join("\n"))
```

This is important code `ApplicationController.render` to render view_component anywhere you want.