DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Action Text Overview

This guide provides you with all you need to get started in handling rich text content.

After reading this guide, you will know:

- How to configure Action Text.
- How to handle rich text content.
- How to style rich text content and attachments.

What is Action Text?

Action Text brings rich text content and editing to Rails. It includes the Trix editor that handles everything from formatting to links to quotes to lists to embedded images and galleries. The rich text content generated by the Trix editor is saved in its own RichText model that's associated with any existing Active Record model in the application. Any embedded images (or other attachments) are automatically stored using Active Storage and associated with the included RichText model.

Trix compared to other rich text editors

Most WYSIWYG editors are wrappers around HTML's `contenteditable` and `execCommand` APIs, designed by Microsoft to support live editing of web pages in Internet Explorer 5.5, and eventually reverse-engineered and copied by other browsers.

Because these APIs were never fully specified or documented, and because WYSIWYG HTML editors are enormous in scope, each browser's implementation has its own set of bugs and quirks, and JavaScript developers are left to resolve the inconsistencies.

Trix sidesteps these inconsistencies by treating contenteditable as an I/O device: when input makes its way to the editor, Trix converts that input into an editing operation on its internal document model, then re-renders that document back into the editor. This gives Trix complete control over what happens after every keystroke, and avoids the need to use execCommand at all.

Installation

Run `bin/rails action_text:install` to add the Yarn package and copy over the necessary migration. Also, you need to set up Active Storage for embedded

images and other attachments. Please refer to the Active Storage Overview guide.

NOTE: Action Text uses polymorphic relationships with the `action_text_rich_texts` table so that it can be shared with all models that have rich text attributes. If your models with Action Text content use UUID values for identifiers, all models that use Action Text attributes will need to use UUID values for their unique identifiers. The generated migration for Action Text will also need to be updated to specify `type: :uuid` for the `:record references` line.

After the installation is complete, a Rails app should have the following changes:

1. Both `trix` and `@rails/actiontext` should be required in your JavaScript entrypoint.

   ```
   // application.js
   import "trix"
   import "@rails/actiontext"
   ```

2. The `trix` stylesheet will be included together with Action Text styles in your `application.css` file.

Creating Rich Text content

Add a rich text field to an existing model:

```
# app/models/message.rb
class Message < ApplicationRecord
  has_rich_text :content
end
```

or add rich text field while creating a new model using:

```
bin/rails generate model Message content:rich_text
```

Note: you don't need to add a `content` field to your `messages` table.

Then use `rich_text_area` to refer to this field in the form for the model:

```
<%# app/views/messages/_form.html.erb %>
<%= form_with model: message do |form| %>
  <div class="field">
    <%= form.label :content %>
    <%= form.rich_text_area :content %>
  </div>
<% end %>
```

And finally, display the sanitized rich text on a page:

```
<%= @message.content %>
```

To accept the rich text content, all you have to do is permit the referenced attribute:

```ruby
class MessagesController < ApplicationController
  def create
    message = Message.create! params.require(:message).permit(:title, :content)
    redirect_to message
  end
end
```

Rendering Rich Text content

By default, Action Text will render rich text content inside an element with the `.trix-content` class:

```
<%# app/views/layouts/action_text/contents/_content.html.erb %>
<div class="trix-content">
  <%= yield %>
</div>
```

Elements with this class, as well as the Action Text editor, are styled by the `trix` stylesheet. To provide your own styles instead, remove the `= require trix` line from the `app/assets/stylesheets/actiontext.css` stylesheet created by the installer.

To customize the HTML rendered around rich text content, edit the `app/views/layouts/action_text/contents/_content.html.erb` layout created by the installer.

To customize the HTML rendered for embedded images and other attachments (known as blobs), edit the `app/views/active_storage/blobs/_blob.html.erb` template created by the installer.

Rendering attachments

In addition to attachments uploaded through Active Storage, Action Text can embed anything that can be resolved by a Signed GlobalID.

Action Text renders embedded `<action-text-attachment>` elements by resolving their `sgid` attribute into an instance. Once resolved, that instance is passed along to `render`. The resulting HTML is embedded as a descendant of the `<action-text-attachment>` element.

For example, consider a `User` model:

```ruby
# app/models/user.rb
class User < ApplicationRecord
  has_one_attached :avatar
end

user = User.find(1)
user.to_global_id.to_s #=> gid://MyRailsApp/User/1
user.to_signed_global_id.to_s #=> BAh7CEkiCG...
```

Next, consider some rich text content that embeds an `<action-text-attachment>` element that references the `User` instance's signed GlobalID:

```
<p>Hello, <action-text-attachment sgid="BAh7CEkiCG..."></action-text-attachment>.</p>
```

Action Text resolves uses the "BAh7CEkiCG..." String to resolve the `User` instance. Next, consider the application's `users/user` partial:

```
<%# app/views/users/_user.html.erb %>
<span><%= image_tag user.avatar %> <%= user.name %></span>
```

The resulting HTML rendered by Action Text would look something like:

```
<p>Hello, <action-text-attachment sgid="BAh7CEkiCG..."><span><img src="..."> Jane Doe</span>
```

To render a different partial, define `User#to_attachable_partial_path`:

```
class User < ApplicationRecord
  def to_attachable_partial_path
    "users/attachable"
  end
end
```

Then declare that partial. The `User` instance will be available as the `user` partial-local variable:

```
<%# app/views/users/_attachable.html.erb %>
<span><%= image_tag user.avatar %> <%= user.name %></span>
```

To integrate with Action Text `<action-text-attachment>` element rendering, a class must:

- include the `ActionText::Attachable` module
- implement `#to_sgid(**options)` (made available through the `GlobalID::Identification` concern)
- (optional) declare `#to_attachable_partial_path`

By default, all `ActiveRecord::Base` descendants mix-in `GlobalID::Identification` concern, and are therefore `ActionText::Attachable` compatible.

Avoid N+1 queries

If you wish to preload the dependent `ActionText::RichText` model, assuming your rich text field is named `content`, you can use the named scope:

```
Message.all.with_rich_text_content # Preload the body without attachments.
Message.all.with_rich_text_content_and_embeds # Preload both body and attachments.
```

API / Backend development

1. A backend API (for example, using JSON) needs a separate endpoint for uploading files that creates an `ActiveStorage::Blob` and returns its

`attachable_sgid`:

```
{
  "attachable_sgid": "BAh7CEkiCG..."
}
```

2. Take that `attachable_sgid` and ask your frontend to insert it in rich text content using an `<action-text-attachment>` tag:

```
<action-text-attachment sgid="BAh7CEkiCG..."></action-text-attachment>
```

This is based on Basecamp, so if you still can't find what you are looking for, check this Basecamp Doc.