In this chapter, you'll see:

- Receiving email with Action Mailbox
- Writing and storing rich text with Action Text
- Managing cloud storage with Active Storage

CHAPTER 16

Task K: Receive Emails and Respond with Rich Text

We've now got a fully functioning store, internationalized for global domination, but what if a customer has a problem they can't solve using our site? With Rails, we can easily do what most e-commerce sites do, which is allow customers to email us so we can solve their problem and write them back with a solution.

Hopefully, you've come to expect by now that Rails has us covered. We sent emails to our customers in Iteration H1: Sending Confirmation Emails, on page 189, but Rails includes a powerful way to *receive* emails called Action Mailbox.[1] We'll learn how that works in this chapter. We'll also learn how to create richly formatted text in our replies by using a rich-text editing system included with Rails called Action Text.[2]

Both Action Text and Action Mailbox rely on another Rails library called Active Storage. Active Storage is an abstraction around cloud storage systems like Amazon's S3. Both incoming emails and rich-text attachments are stored in the cloud using Active Storage. We'll explain why as we go.

Iteration K1: Receiving Support Emails with Action Mailbox

Configuring Rails to receive emails requires three steps: initially setting up Action Mailbox, setting up Active Storage to hold the raw emails we receive, and implementing a *mailbox*, which is like a controller that handles incoming emails.

1. https://guides.rubyonrails.org/action_mailbox_basics.html
2. https://guides.rubyonrails.org/action_text_overview.html

Setting up Action Mailbox

To set up Action Mailbox in our app, we'll run a Rake task that will create some configuration files, a base mailbox class we'll inherit from, and some database tables that Rails will use to store information about incoming emails. Let's run the Rake task:

```
> bin/rails action_mailbox:install
Copying application_mailbox.rb to app/mailboxes
      create  app/mailboxes/application_mailbox.rb
Copied migration
  20221207000011_create_active_storage_tables.active_storage.rb
    from active_storage
Copied migration
  20221207000012_create_action_mailbox_tables.action_mailbox.rb
    from action_mailbox
```

Note that a) we've reformatted our output to fit the pages in the book and b) since there were two migrations created and migration filenames have a date and timestamp in them, your filenames won't exactly match ours. Next, we'll add the tables that Rake task created to our development and test databases:

```
> bin/rails db:migrate
== 20221207191846 CreateActiveStorageTables: migrating ======================
-- create_table(:active_storage_blobs, {})
   -> 0.0015s
-- create_table(:active_storage_attachments, {})
   -> 0.0013s
== 20221207191846 CreateActiveStorageTables: migrated (0.0029s) =============

== 20221207191847 CreateActionMailboxTables: migrating ======================
-- create_table(:action_mailbox_inbound_emails)
   -> 0.0017s
== 20221207191847 CreateActionMailboxTables: migrated (0.0017s) =============
```

In the real world, we'd also need to configure Action Mailbox for our particular incoming email service provider. The Rails Guide[3] is the best place to look for how to do that. We won't set one up here since setting up accounts with services like Amazon SES or Mailgun is somewhat involved (though once you have your account set up, configuring Rails to use it is a snap). For our immediate needs, Rails provides a way to simulate sending emails, which we'll see in a moment.

The way Action Mailbox works is that all incoming emails get stored in a cloud storage system like Amazon's S3. Rails includes a library called Active Storage

3. https://guides.rubyonrails.org/action_mailbox_basics.html#configuration

that abstracts away the details of the cloud service you're using. We'll need to configure Active Storage for Action Mailbox to work properly.

Setting up Active Storage

As with your real-world email provider, your real-world cloud storage provider will require specific configuration in Rails, and the Guide[4] can give you the details. For our purposes, we'll set up the disk-based service that works with our local disk. This will allow us to fully use Active Storage locally, which means Action Mailbox can work locally.

To set that up, we'll need to configure the service in our app's configuration and then tell Rails where to store the files that Active Storage will manage.

First, edit config/environments/development.rb, adding this line to the configuration at the end of the block:

rails7/depot_ua/config/environments/development.rb

```
  config.active_storage.service = :local
end
```

We will explain what :local means in a moment. Now, add a similar line to config/environments/test.rb but using the :test service instead:

rails7/depot_ua/config/environments/test.rb

```
  config.active_storage.service = :test
end
```

With those added, we must now define what those symbols mean by creating config/storage.yml to look like so:

rails7/depot_ua/config/storage.yml

```
local:
  service: Disk
  root: <%= Rails.root.join("storage") %>

test:
  service: Disk
  root: <%= Rails.root.join("tmp/storage") %>
```

The root key in this file should match the values we used in the files in config/environments. In this case, we've configured both :local and :test to use Active Storage's disk-based service, with our development environment (:local) using the directory storage that's in the root of our project and the test environment (:test) using tmp/storage.

4. https://guides.rubyonrails.org/active_storage_overview.html

With this set up, when we receive an email, the entire payload gets written to our storage service and, as we'll see in a moment, we can access parts of that email to trigger whatever logic we need in our Rails app. The reason Rails does this is that emails can be large (especially if they have attachments), and you don't necessarily want to store very large objects in a relational database. It's much more common to store such data to disk or with a cloud storage provider and store a reference to that object in the database.

Now that we've done the one-time setup, let's create a mailbox to receive our support request emails from customers.

Creating a Mailbox to Receive Emails

Action Mailbox works by routing incoming emails to a mailbox. A mailbox is a subclass of ApplicationMailbox with a method named process() that is called for each email routed to that mailbox. The way emails get routed is similar to how web requests get routed in config/routes.rb. For email, you'll tell Rails what sorts of emails you want routed where.

We want emails to support@example.com to get routed to a mailbox so we can handle them. The way to do that is to insert a call to the method routing() inside ApplicationMailbox, like so:

rails7/depot_ua/app/mailboxes/application_mailbox.rb

```
class ApplicationMailbox < ActionMailbox::Base
  routing "support@example.com" => :support
end
```

This tells Rails that any email to (or cc'd to) support@example.com should be handled by the class SupportMailbox. We can create that class using a Rails generator like so:

```
> bin/rails generate mailbox support
      create  app/mailboxes/support_mailbox.rb
      invoke  test_unit
      create    test/mailboxes/support_mailbox_test.rb
```

If you look at app/mailboxes/support_mailbox.rb, you'll see a few lines of code, notably an empty method called process():

```
class SupportMailbox < ApplicationMailbox
  def process
  end
end
```

Now, every email we receive at support@example.com will trigger a call to process() in SupportMailbox. Inside the process() method, we have access to the special variable mail. This is an instance of Mail::Message[5] and allows us to access the various bits of an email you might expect to have, such as who sent it, the subject, and the contents.

Let's see how this works before getting too far along by adding some puts() calls into our mailbox:

rails7/depot_ua/app/mailboxes/support_mailbox.rb

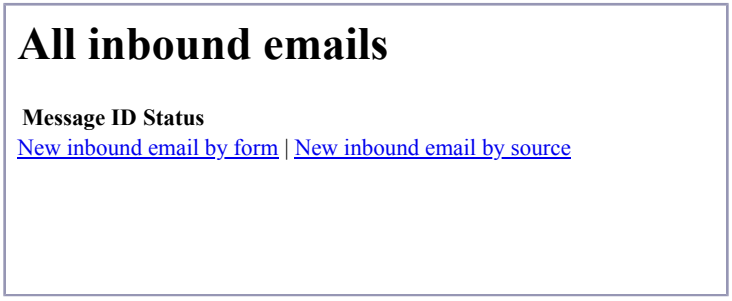
```
class SupportMailbox < ApplicationMailbox
  def process
    puts "START SupportMailbox#process:"
    puts "From    : #{mail.from_address}"
    puts "Subject: #{mail.subject}"
    puts "Body    : #{mail.body}"
    puts "END SupportMailbox#process:"
  end
end
```

Since we didn't configure a real email provider, how do we trigger our mailbox locally? The answer is a special UI included with Rails called a *conductor*.

Using the Conductor to Send Emails Locally

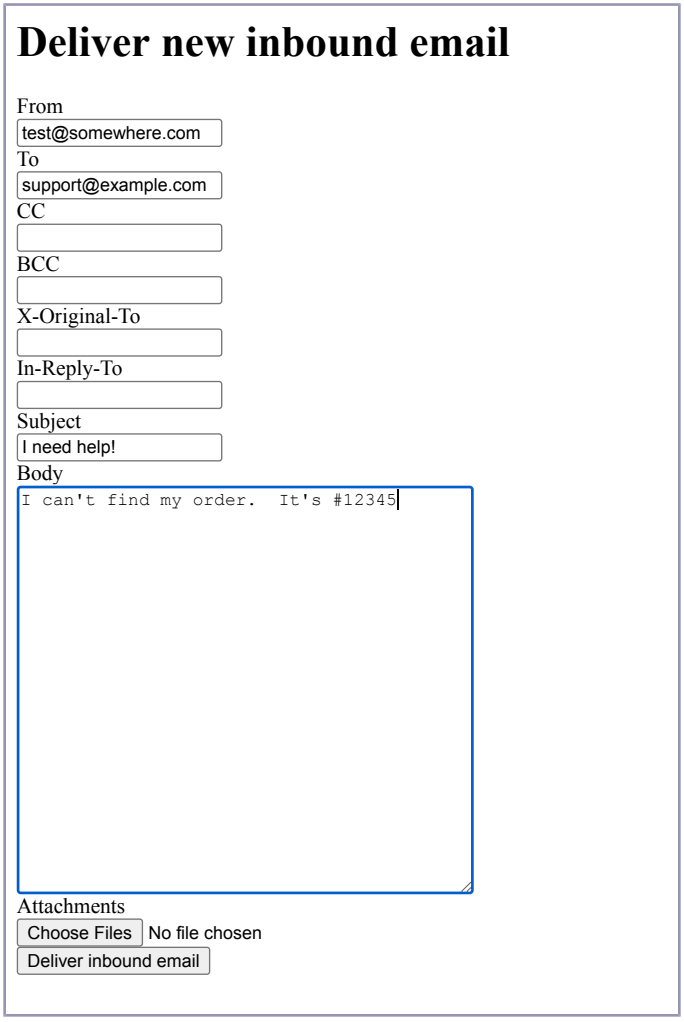
Action Mailbox includes a special developer-only UI we can use to send emails to ourselves. This allows us to see our mailbox working end-to-end without having to configure a real email provider. To see it, start up your server (or restart it if it's already running).

Navigate to http://localhost:3000/rails/conductor/action_mailbox/inbound_emails and you should see a bare-bones UI that includes a link labeled "New inbound email by form", like so:



5. https://www.rubydoc.info/github/mikel/mail/Mail/Message

Click that link, and you should see a very basic UI to write an email, like so:



Fill this in, remembering to use support@example.com as the From email so that the email gets routed to your mailbox. If you then click Deliver inbound email, and flip back to where you ran your server, you should see, among other log output, the puts() you inserted:

```
START SupportMailbox#process:
From    : test@somewhere.com
Subject: I need help!
Body    : I can't find my order.  It's #12345
END SupportMailbox#process:
```

Now that we see how all the parts fit together, let's write the real code to store the request for help from the customer (as well as how to test our mailbox with a unit test).

Iteration K2: Storing Support Requests from Our Mailbox

As we mentioned above, the purpose of mailboxes is to allow us to execute code on every email we receive. Because emails come in whenever the sender sends them, we'll need to store the details of a customer support request somewhere for an administrator to handle later. To that end, we'll create a new model called SupportRequest that will hold the relevant details of the request, and have the process() method of SupportMailbox create an instance for each email we get (in the final section of this chapter we'll display these in a UI so an admin can respond).

Creating a Model for Support Requests

We want our model to hold the sender's email, the subject and body of the email, and a reference to the customer's most recent order if there's one on file. First, let's create the model using a Rails generator:

```
> bin/rails g model support_request
      invoke  active_record
      create    db/migrate/20221207000013_create_support_requests.rb
      create    app/models/support_request.rb
      invoke    test_unit
      create      test/models/support_request_test.rb
      create      test/fixtures/support_requests.yml
```

This created a migration for us, which is currently empty (remember that migration filenames have a date and time in them, so your filename will be slightly different). Let's fill that in.

rails7/depot_tb/db/migrate/20221207000013_create_support_requests.rb

```
class CreateSupportRequests < ActiveRecord::Migration[7.0]
  def change
    create_table :support_requests do |t|
➤     t.string :email, comment: "Email of the submitter"
➤     t.string :subject, comment: "Subject of their support email"
➤     t.text :body, comment: "Body of their support email"
➤     t.references :order,
➤                  foreign_key: true,
➤                  comment: "their most recent order, if applicable"
      t.timestamps
    end
  end
end
```

With this in place, we can create this table via bin/rails db:migrate:

```
> bin/rails db:migrate
== 20221207121503 CreateSupportRequests: migrating ============================
-- create_table(:support_requests)
   -> 0.0016s
== 20221207121503 CreateSupportRequests: migrated (0.0017s) ===================
```

We'll also need to adjust the model itself to optionally reference an order:

rails7/depot_tb/app/models/support_request.rb

```
class SupportRequest < ApplicationRecord
  belongs_to :order, optional: true
end
```

Now, we can create instances of SupportRequest from our mailbox.

Creating Support Requests from Our Mailbox

Our mailbox needs to do two things. First, it needs to create an instance of SupportRequest for each email that comes in. But it also needs to connect that request to the user's most recent order if there's one in our database (this will allow our admin to quickly reference the order that might be causing trouble).

As you recall, all orders have an email associated with them. So to get the most recent order for an email, we can use where() to search all orders by email, order() to order the results by the create data, and first() to grab the most recent one. With that, we can use the methods on mail we saw earlier to create the SupportRequest.

Here's the code we need in app/mailboxes/support_mailbox.rb (which replaces the calls to puts() we added before):

rails7/depot_tb/app/mailboxes/support_mailbox.rb

```
class SupportMailbox < ApplicationMailbox
  def process
    recent_order = Order.where(email: mail.from_address.to_s).
                         order('created_at desc').
                         first
    SupportRequest.create!(
      email: mail.from_address.to_s,
      subject: mail.subject,
      body: mail.body.to_s,
      order: recent_order
    )
  end
end
```

Why Don't We Access Emails Directly When Needed?

It might seem easier to simply access the customer emails whenever we need them rather than pluck out the data we want and store it into a database. There are two reasons not to do this.

The first, and most practical reason, is about separation of concerns. Our support requests only need part of what is in the emails, but they also might need more metadata than the customer sends us. To keep our code organized and clean, it's better to store what we need explicitly.

The second reason is one of Rails' famously held opinions. Rails arranges for all emails to be deleted after thirty days. The reasoning is that emails contain personal data that we don't want to hold onto unnecessarily.

Protecting the personal data of your customers is a good practice, and it's one that's more and more required by law. For example, the European General Data Protection Regularly (GDPR) requires, among other things, that you delete any personal data you have within one month of a request to do so. By auto-deleting personal data every thirty days, you automatically comply with this requirement.[a]

a. We're not lawyers, so please don't take this sidebar as legal advice!

Now, restart your server and navigate to the conductor at http://localhost:3000/rails/conductor/action_mailbox/inbound_emails. Click Deliver new inbound email and send another email (remember to send it to support@example.com).

Now, quit your server and start up the Rails console. This will allow us to check that a new SupportRequest was created (remember we have to format this to fit in the book, so your output will be on fewer, longer lines):

```
> bin/rails console
irb(main):001:0> SupportRequest.first
   (1.5ms)  SELECT sqlite_version(*)
  SupportRequest Load (0.1ms)
    SELECT "support_requests".* FROM "support_requests"
      ORDER BY "support_requests"."id" ASC LIMIT ?  [["LIMIT", 1]]
=> #<SupportRequest
       id: 1,
       email: "chris@somewhere.com",
       subject: "Missing book!",
       body: "I can't find my book that I ordered.  Please help!",
       order_id: nil,
       created_at: "2021-01-19 12:29:17",
       updated_at: "2021-01-19 12:29:17">
```

You should see the data you entered into the conductor saved in the SupportRequest instance. You can also try this using the email of an order you have in your system to verify it locates the most recent order. Of course, manually checking our code isn't ideal. We would like to have an automated test. Fortunately, Rails provides a simple way to test our mailboxes, which we'll learn about now.

Testing Our Mailbox

When we used the generator to create our mailbox, you probably noticed the file test/mailboxes/support_mailbox_test.rb get created. This is where we'll write our test. Since we generally know how to write tests, all we need to know now is how to trigger an email. Action Mailbox provides the method receive_inbound_email_from_mail() which we can use in our tests to do just that.

We need two tests to cover the functionality of our mailbox. The first is to send an email from a customer without an order and verify we created a SupportRequest instance. The second is to send an email from a customer who *does* have orders and verify that the SupportRequest instance is correctly connected to their most recent order.

The first test is most straightforward since we don't need any test setup, so we'll create a new test() block inside test/mailboxes/support_mailbox_test.rb, like so:

rails7/depot_tb/test/mailboxes/support_mailbox_test.rb

```
require "test_helper"

class SupportMailboxTest < ActionMailbox::TestCase
  test "we create a SupportRequest when we get a support email" do
    receive_inbound_email_from_mail(
      to: "support@example.com",
      from: "chris@somewhere.net",
      subject: "Need help",
      body: "I can't figure out how to check out!!"
    )

    support_request = SupportRequest.last
    assert_equal "chris@somewhere.net", support_request.email
    assert_equal "Need help", support_request.subject
    assert_equal "I can't figure out how to check out!!",
      support_request.body
    assert_nil support_request.order
  end

end
```

If we run this test now, it should pass:

```
> bin/rails test test/mailboxes/support_mailbox_test.rb
Run options: --seed 26908

# Running:

.

Finished in 0.322222s, 3.1035 runs/s, 12.4138 assertions/s.
1 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

Great! For the second test, we'll need to create a few orders before we send the email. You'll recall from Test Fixtures, on page 92, that we can use fixtures to set up test data in advance. We have one we can use already, but ideally we'd have a total of two orders for the user sending the email and a third order from another user. That would validate that we're both searching for the right user *and* selecting the most recent order.

Let's add two new fixtures to test/fixtures/orders.yml

rails7/depot_tb/test/fixtures/orders.yml

```
# Read about fixtures at
# https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
  name: Dave Thomas
  address: MyText
  email: dave@example.org
  pay_type: Check

another_one:
  name: Dave Thomas
  address: 123 Any St
  email: dave@example.org
  pay_type: Check
  created_at: <%= 2.days.ago %>

other_customer:
  name: Chris Jones
  address: 456 Somewhere Ln
  email: chris@nowhere.net
  pay_type: Check
two:
  name: MyString
  address: MyText
  email: MyString
  pay_type: 1
```

Note how we're using ERB inside our fixture. This code is executed when we request a fixture and we're using it to force an older creation date for one of our orders. By default, Rails sets created_at on models it creates from fixtures to the current time. When we ask Rails to create that particular fixture with orders(:another_one), it will execute the code inside the <%= and %>, effectively setting the created_at value to the date as of two days ago.

With these fixtures available, we can write our second test, like so:

rails7/depot_tb/test/mailboxes/support_mailbox_test.rb

```
require "test_helper"

class SupportMailboxTest < ActionMailbox::TestCase

  # previous test

  test "we create a SupportRequest with the most recent order" do
    recent_order  = orders(:one)
    older_order   = orders(:another_one)
    non_customer  = orders(:other_customer)

    receive_inbound_email_from_mail(
      to: "support@example.com",
      from: recent_order.email,
      subject: "Need help",
      body: "I can't figure out how to check out!!"
    )

     support_request = SupportRequest.last
     assert_equal recent_order.email, support_request.email
     assert_equal "Need help", support_request.subject
     assert_equal "I can't figure out how to check out!!", support_request.body
     assert_equal recent_order, support_request.order
  end

end
```

Next, rerun the test and we should see our new test is passing:

```
> bin/rails test test/mailboxes/support_mailbox_test.rb
Run options: --seed 47513

# Running:

..

Finished in 0.384217s, 5.2054 runs/s, 20.8216 assertions/s.
2 runs, 8 assertions, 0 failures, 0 errors, 0 skips
```

Nice! We can now confidently write code to handle incoming emails and test it with an automated test. Now what do we do with these SupportRequest instances we're creating? We'd like to allow an administrator to respond to them. We

could do that with plain text, but let's learn about another part of Rails called Action Text that will allow us to author rich text we can use to respond.

Iteration K3: Responding with Rich Text

To allow our admins to respond to support requests, we'll need to make a new UI for them to see the requests that need a response, a way for them to provide a response, and then some code to email the customer back. We know how to do all of these things, but this is a great opportunity to learn about Action Text, which is a Rails library that allows us to easily provide a rich-text editing experience. We can use this to allow our admins to write a fully formatted response and not just plain text.

Let's first quickly create the UI where we'll see the support requests and edit them. This should be old hat for you by now, so we'll go quickly. Add a new route to config/routes.rb for the index() and update() methods:

rails7/depot_tb/config/routes.rb

```
Rails.application.routes.draw do
  get 'admin' => 'admin#index'
  controller :sessions do
    get  'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end
  get 'sessions/create'
  get 'sessions/destroy'

  # START_HIGLIGHT
  resources :support_requests, only: %i[ index update ]
  # END_HIGLIGHT
  resources :users
  resources :products do
    get :who_bought, on: :member
  end

  scope '(:locale)' do
    resources :orders
    resources :line_items
    resources :carts
    root 'store#index', as: 'store_index', via: :all
  end
end
```

Now, create app/controllers/support_requests_controller.rb and implement index(), like so (we'll see update() in a moment):

rails7/depot_tb/app/controllers/support_requests_controller.rb

```
class SupportRequestsController < ApplicationController
  def index
    @support_requests = SupportRequest.all
  end

end
```

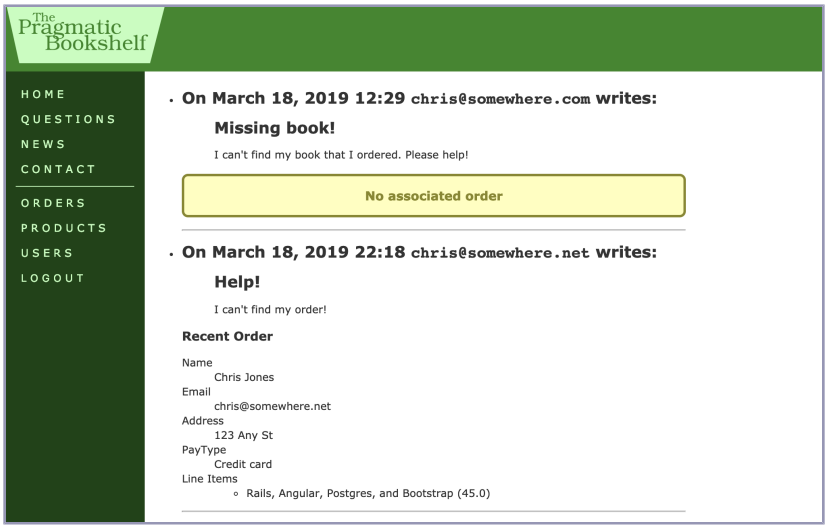
Next, we'll create the view in app/views/support_requests/index.html.erb:

rails7/depot_tb/app/views/support_requests/index.html.erb

```
<ul>
  <% @support_requests.each do |support_request| %>
    <li>
      <h1>
        On <%= support_request.created_at.to_formatted_s(:long) %>
        <code><%= support_request.email %></code> writes:
      </h1>
      <p>
        <blockquote>
          <h2><%= support_request.subject %></h2>
          <%= support_request.body %>
        </blockquote>
      </p>
      <% if support_request.order %>
        <h3>Recent Order</h3>
        <dl>
          <dt>Name</dt>
          <dd><%= support_request.order.name %></dd>

          <dt>Email</dt>
          <dd><%= support_request.order.email %></dd>

          <dt>Address</dt>
          <dd><%= support_request.order.address %></dd>

          <dt>PayType</dt>
          <dd><%= support_request.order.pay_type %></dd>

          <dt>Line Items</dt>
          <dd>
            <ul>
              <% support_request.order.line_items.each do |line_item| %>
                <li>
                  <%= line_item.product.title %>
                  (<%= line_item.product.price %>)
                </li>
              <% end %>
            </ul>
          </dd>
        </dl>
      <% else %>
```

```
          <h3 class="notice">No associated order</h3>
        <% end %>
        <hr>
      </li>
    <% end %>
</ul>
```

Restart your server, create a few orders and, using the Rails conductor we saw earlier, create a few support tickets. Be sure at least one of them is from an email you used to create an order. When you've done that, navigate to http://localhost:3000/admin and log in. Once you've done *that*, navigate to http://localhost:3000/support_requests and you should see the UI you just created with your support requests rendered:



It's not pretty, but it'll work for now. Next, we need to add the ability to write a response. If we were OK with plain text, we would make a new attribute on SupportRequest to hold the response, then wire up a form to write it, just like we've done several times. With rich text, it works a bit differently.

Action Text stores the rich text in its own table outside of the model's. In our SupportRequest model, we'll tell Rails that we have a rich-text field that we want Action Text to manage by using the has_rich_text() method, like so:

rails7/depot_tc/app/models/support_request.rb

```
class SupportRequest < ApplicationRecord

  belongs_to :order, optional: true

    has_rich_text :response
end
```

This method (and the rest of Action Text) won't work without some setup, which we can do with the Rake task action_text:install:

```
> bin/rails action_text:install
      append  app/javascript/application.js
      append  config/importmap.rb
      create  app/assets/stylesheets/actiontext.css
      append  app/assets/stylesheets/application.tailwind.css
      create  app/views/active_storage/blobs/_blob.html.erb
      create  app/views/layouts/action_text/contents/_content.html.erb
Ensure image_processing gem has been enabled so image uploads will work
(remember to bundle!)
        gsub  Gemfile
       rails  railties:install:migrations FROM=active_storage,action_text
Copied migration 20221207145849_create_action_text_tables.action_text.rb
 from action_text
      invoke  test_unit
      create    test/fixtures/action_text/rich_texts.yml
```

You'll notice that the generator created a database migration. This is for the tables that Action Text uses to store the rich text itself.

Let's add those by running the db:migrate task:

```
> bin/rails db:migrate
== 20221207120454 CreateActionTextTables: migrating ======================
-- create_table(:action_text_rich_texts, {:id=>:primary_key})
   -> 0.0016s
== 20221207120454 CreateActionTextTables: migrated (0.0017s) =============
```

With all of that back-end setup out of the way, we can now make our UI. We will create this in the exact same way we've created other forms in our app, with the exception of the text area. Instead of using the text_area() form helper to make a regular HTML textarea tag, we'll use rich_text_area(), which will set up the Trix editor for us, enabling the UI part of Action Text.

Add this to app/views/support_requests/index.html.erb:

rails7/depot_tc/app/views/support_requests/index.html.erb

```
<ul>
  <% @support_requests.each do |support_request| %>
    <li>
      <h1>
        On <%= support_request.created_at.to_formatted_s(:long) %>
        <code><%= support_request.email %></code> writes:
      </h1>
      <p>
        <blockquote>
          <h2><%= support_request.subject %></h2>
```

```
      <%= support_request.body %>
    </blockquote>
  </p>
  <% if support_request.order %>
    <h3>Recent Order</h3>
    <dl>
      <dt>Name</dt>
      <dd><%= support_request.order.name %></dd>

      <dt>Email</dt>
      <dd><%= support_request.order.email %></dd>

      <dt>Address</dt>
      <dd><%= support_request.order.address %></dd>

      <dt>PayType</dt>
      <dd><%= support_request.order.pay_type %></dd>

      <dt>Line Items</dt>
      <dd>
        <ul>
          <% support_request.order.line_items.each do |line_item| %>
            <li>
              <%= line_item.product.title %>
              (<%= line_item.product.price %>)
            </li>
          <% end %>
        </ul>
      </dd>
    </dl>
  <% else %>
    <h3 class="notice">No associated order</h3>
  <% end %>
➤ <% if support_request.response.blank? %>
➤   <%= form_with(model: support_request,
➤                 local: true,
➤                 class: "depot_form") do |form| %>
➤     <div class="field">
➤       <%= form.label :response, "Write Response" %>
➤       <%= form.rich_text_area :response, id: :support_request_response %>
➤     </div>
➤     <div class="actions">
➤       <%= form.submit "Send Response" %>
➤     </div>
➤   <% end %>
➤ <% else %>
➤   <h4>Our response:</h4>
➤   <p>
➤     <blockquote>
➤       <%= support_request.response %>
➤     </blockquote>
➤   </p>
```

```
      <% end %>

      <hr>
    </li>
  <% end %>
</ul>
```

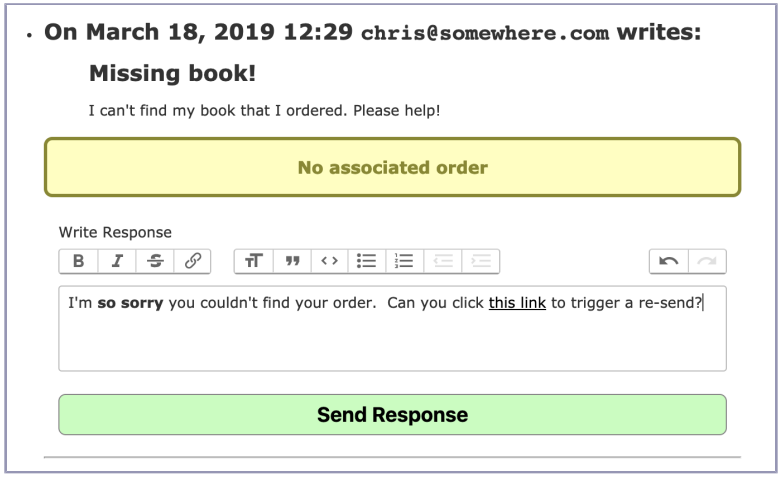
Note that we check to see if the support request has a response, and if it does, we render it. As we'll see, this has been enhanced by Action Text.

The last step is to implement update() in our controller:

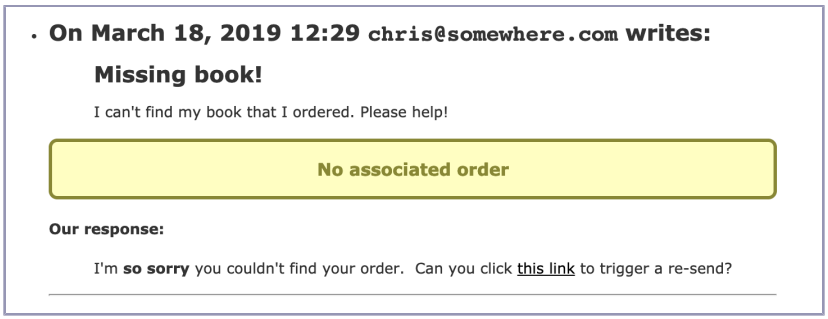
rails7/depot_tc/app/controllers/support_requests_controller.rb

```
class SupportRequestsController < ApplicationController
  def index
    @support_requests = SupportRequest.all
  end

  def update
    support_request = SupportRequest.find(params[:id])
    support_request.update(response: params.require(:support_request)[:response])
    SupportRequestMailer.respond(support_request).deliver_now
    redirect_to support_requests_path
  end
end
```

Now, start up your server and, assuming you've created some support requests, you should now see a rich-text editor instead of a plain old text area, like so:



You can see in the screenshot on page 264 that we've added rich text to the text area using the editor's controls. Try that in your environment, then click Send Response. The page will refresh and, because we've now saved a response with this SupportRequest, you'll see the rich text rendered…in rich text!



We learned how to send emails in Chapter 13, Task H: Sending Emails, on page 189, but when dealing with rich text and the need to send a plain-text email, we have to strip out the rich text. So let's set up the mailer to respond to the user and, when we create the plain-text template, we'll see how to strip out the rich text. We'll start this off by creating the mailer using the Rails generator:

```
> bin/rails generate mailer support_request respond
      create  app/mailers/support_request_mailer.rb
      invoke  erb
      create    app/views/support_request_mailer
      create    app/views/support_request_mailer/respond.text.erb
      create    app/views/support_request_mailer/respond.html.erb
      invoke  test_unit
      create    test/mailers/support_request_mailer_test.rb
      create    test/mailers/previews/support_request_mailer_preview.rb
```

Our mailer will look similar to the mailers we've created in the past. This is what your app/mailers/support_request_mailer.rb should look like:

rails7/depot_tc/app/mailers/support_request_mailer.rb

```
class SupportRequestMailer < ApplicationMailer

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.support_request_mailer.respond.subject
  #

➤   default from: "support@example.com"
➤
➤   def respond(support_request)
➤     @support_request = support_request
➤     mail to: @support_request.email, subject: "Re: #{@support_request.subject}"
➤   end

end
```

For the views, we'll show our response and quote the user's original email. As we saw in our web view, Rails will handle rendering the rich text for us, so the HTML mail view in app/views/support_request_mailer/respond.html.erb will look fairly straightforward:

rails7/depot_tc/app/views/support_request_mailer/respond.html.erb

```
<%= @support_request.response %>
<hr>
<blockquote>
  <%= @support_request.body %>
</blockquote>
```

We also want to include a plain-text version, since not everyone wants rich text in their emails. In the case of a plain-text email, we want to strip out the rich text from our response. Action Text provides a method to do that, called to_plain_text(), which we can use in app/views/support_request_mailer/respond.text.erb, like so:

rails7/depot_tc/app/views/support_request_mailer/respond.text.erb

```
<%= @support_request.response.to_plain_text %>

---

<%= @support_request.body %>
```

The last step is to add a call to our mailer when we update the SupportRequest:

```
class SupportRequestsController < ApplicationController
  def index
    @support_requests = SupportRequest.all
  end

  def update
    support_request = SupportRequest.find(params[:id])
    support_request.update(response: params.require(:support_request)[:response])
➤   SupportRequestMailer.respond(support_request).deliver_now
    redirect_to support_requests_path
  end
end
```

Now, if you restart your server and respond to a support request, you'll see the mail printed out in your log, and you should see that the plain-text part of the email is free of formatting.

What We Just Did

- We configured and set up Action Mailbox to allow our app to receive support emails. We saw how to configure Rails to inspect each incoming email and route it to the right bit of code, called a mailbox.

- We also configured Active Storage, which Rails uses to store the raw emails it processes. With it set up, we could easily access cloud storage for any other purpose we might need.
- We used Action Text to enable rich-text editing for responding to support requests. With just a few lines of code, we have a cross-browser rich-text editing experience that works.
- We stripped out the rich text to send a plain-text email of our rich-text response.

Playtime

Here are some things you can try on your own:

- Modify the product editor to allow products to have rich text.
- Change the support request to find all orders for the email, not just the most recent.