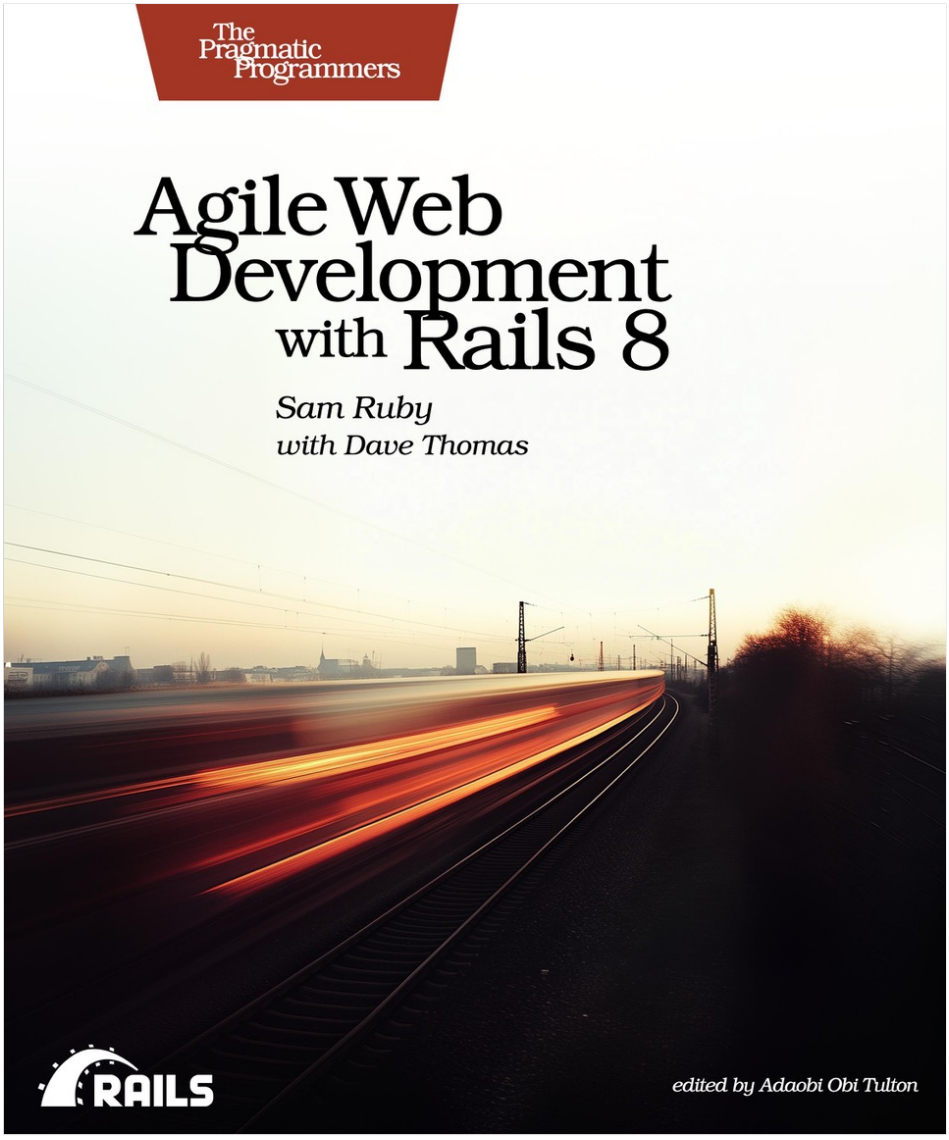

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

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 generate model support_request
      invoke  active_record
      create    db/migrate/20241021000014_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.

rails80/depot_tb/db/migrate/20241021000014_create_support_requests.rb

```
class CreateSupportRequests < ActiveRecord::Migration[8.0]
  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
== 20241021121503 CreateSupportRequests: migrating ============================
-- create_table(:support_requests)
   -> 0.0016s
== 20241021121503 CreateSupportRequests: migrated (0.0017s) ===================
```

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

rails80/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):

rails80/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:

rails80/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 ?, 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

rails80/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:

rails80/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.