In this chapter, you'll see:

- Sending email
- Running background code with Active Job
- System testing background jobs and email

CHAPTER 13

Task H: Sending Emails and Processing Payments Efficiently

At this point, we have a website that responds to requests and provides feeds that allow sales of individual titles to be checked periodically. The customer is happier but still not satisfied. The first bit of feedback is that users aren't getting confirmation emails of their purchases. The second is around payment processing. The customer has arranged for us to integrate with a payment processor that can handle all forms of payment we want to support, but the processor's API is very slow. The customer wants to know if that will slow down the site.

Sending email is a common need for any web application, and Rails has you covered via Action Mailer,[1] which you'll learn in this chapter. Dealing with the slow payment-processing API requires learning about the library Action Mailer is built on, Active Job.[2] Active Job allows you to run code in a background process so that the user doesn't have to wait for it to complete. Sending email is slow, which is why Action Mailer uses Active Job to offload the work. This is a common technique you'll use often when developing web applications. Let's take it one step at a time and learn how to send email.

Iteration H1: Sending Confirmation Emails

Sending email in Rails has three basic parts: configuring how email is to be sent, determining when to send the email, and specifying what you want to say. We'll cover each of these three in turn.

1. http://guides.rubyonrails.org/action_mailer_basics.html
2. http://guides.rubyonrails.org/active_job_basics.html

Configuring Email

Email configuration is part of a Rails application's environment and involves a Rails.application.configure block. If you want to use the same configuration for development, testing, and production, add the configuration to environment.rb in the config directory; otherwise, add different configurations to the appropriate files in the config/environments directory.

Inside the block, you need to have one or more statements. You first have to decide how you want mail delivered:

```
config.action_mailer.delivery_method = :smtp
```

Alternatives to :smtp include :sendmail and :test.

The :smtp and :sendmail options are used when you want Action Mailer to attempt to deliver email. You'll clearly want to use one of these methods in production.

The :test setting is great for unit and functional testing, which we'll make use of in Testing Email, on page 195. Email won't be delivered; instead, it'll be appended to an array (accessible via the ActionMailer::Base.deliveries attribute). This is the default delivery method in the test environment. Interestingly, though, the default in development mode is :smtp. If you want Rails to deliver email during the development of your application, this is good. If you'd rather disable email delivery in development mode, edit the development.rb file in the config/environments directory and add the following lines:

```
Rails.application.configure do
  config.action_mailer.delivery_method = :test
end
```

The :sendmail setting delegates mail delivery to your local system's sendmail program, which is assumed to be in /usr/sbin. This delivery mechanism isn't particularly portable, because sendmail isn't always installed in this directory for every operating system. It also relies on your local sendmail supporting the -i and -t command options.

You achieve more portability by leaving this option at its default value of :smtp. If you do so, you'll need also to specify some additional configuration to tell Action Mailer where to find an SMTP server to handle your outgoing email. This can be the machine running your web application, or it can be a separate box (perhaps at your ISP if you're running Rails in a noncorporate environment). Your system administrator will be able to give you the settings for these parameters. You may also be able to determine them from your own mail client's configuration.

The following are typical settings for Gmail: adapt them as you need.

```
Rails.application.configure do
  config.action_mailer.delivery_method = :smtp

  config.action_mailer.smtp_settings = {
    address:        "smtp.gmail.com",
    port:           587,
    domain:         "domain.of.sender.net",
    authentication: "plain",
    user_name:      "dave",
    password:       "secret",
    enable_starttls_auto: true
  }
end
```

As with all configuration changes, you'll need to restart your application if you make changes to any of the environment files.

Sending Email

Now that we have everything configured, let's write some code to send emails.

By now you shouldn't be surprised that Rails has a generator script to create *mailers*. In Rails, a mailer is a class that's stored in the app/mailers directory. It contains one or more methods, with each method corresponding to an email template. To create the body of the email, these methods in turn use views (in the same way that controller actions use views to create HTML and XML). So let's create a mailer for our store application. We'll use it to send two different types of email: one when an order is placed and a second when the order ships. The rails generate mailer command takes the name of the mailer class along with the names of the email action methods:

```
depot> bin/rails generate mailer Order received shipped
    create  app/mailers/order_mailer.rb
    invoke  tailwindcss
    create    app/views/order_mailer
    create    app/views/order_mailer/received.text.erb
    create    app/views/order_mailer/received.html.erb
    create    app/views/order_mailer/shipped.text.erb
    create    app/views/order_mailer/shipped.html.erb
    invoke  test_unit
    create    test/mailers/order_mailer_test.rb
    create    test/mailers/previews/order_mailer_preview.rb
```

Notice that we create an OrderMailer class in app/mailers and two template files, one for each email type, in app/views/order. (We also create a test file; we'll look into this in Testing Email, on page 195.)

Each method in the mailer class is responsible for setting up the environment for sending an email. Let's look at an example before going into detail. Here's the code that was generated for our OrderMailer class, with one default changed:

rails7/depot_q/app/mailers/order_mailer.rb

```
class OrderMailer < ApplicationMailer
  default from: 'Sam Ruby <depot@example.com>'

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.order_mailer.received.subject
  #
  def received
    @greeting = "Hi"

    mail to: "to@example.org"
  end

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.order_mailer.shipped.subject
  #
  def shipped
    @greeting = "Hi"

    mail to: "to@example.org"
  end
end
```

If you're thinking to yourself that this looks like a controller, that's because it does. It includes one method per action. Instead of a call to render(), there's a call to mail(). This method accepts a number of parameters including :to (as shown), :cc, :from, and :subject, each of which does pretty much what you'd expect it to do. Values that are common to all mail() calls in the mailer can be set as defaults by simply calling default, as is done for :from at the top of this class. Feel free to tailor this to your needs.

The comments in this class also indicate that subject lines are already enabled for translation, a subject we'll cover in Chapter 15, Task J: Internationalization, on page 225. For now, we'll simply use the :subject parameter.

As with controllers, templates contain the text to be sent, and controllers and mailers can provide values to be inserted into those templates via instance variables.

Email Templates

The generate script created two email templates in app/views/order_mailer, one for each action in the OrderMailer class. These are regular .erb files. We'll use them to create plain-text emails (you'll see later how to create HTML email). As with the templates we use to create our application's web pages, the files contain a combination of static text and dynamic content. We can customize the template in received.text.erb; this is the email that's sent to confirm an order:

rails7/depot_q/app/views/order_mailer/received.text.erb

```
Dear <%= @order.name %>

Thank you for your recent order from The Pragmatic Store.

You ordered the following items:

<%= render @order.line_items -%>

We'll send you a separate e-mail when your order ships.
```

The partial template that renders a line item formats a single line with the item quantity and the title. Because we're in a template, all the regular helper methods, such as truncate(), are available:

rails7/depot_q/app/views/line_items/_line_item.text.erb

```
<%= sprintf("%2d x %s",
            line_item.quantity,
            truncate(line_item.product.title, length: 50)) %>
```

We now have to go back and fill in the received() method in the OrderMailer class:

rails7/depot_qa/app/mailers/order_mailer.rb

```
def received(order)
  @order = order

  mail to: order.email, subject: 'Pragmatic Store Order Confirmation'
end
```

What we did here is add order as an argument to the method-received call, add code to copy the parameter passed into an instance variable, and update the call to mail() specifying where to send the email and what subject line to use.

Generating Emails

Now that we have our template set up and our mailer method defined, we can use them in our regular controllers to create and/or send emails. Note that just calling the method we defined isn't enough; we also need to tell Rails to actually send the email. The reason this doesn't happen automatically is that Rails can't be 100 percent sure if you want to deliver the email right this moment, while the user waits, or later, in a background job.

Generally, you don't want the user to have to wait for emails to get sent, because this can take a while. Instead, we'll send it in a background job (which we'll learn more about later in the chapter) by calling deliver_later() (to send the email right now, you'd use deliver_now().[3])

rails7/depot_qa/app/controllers/orders_controller.rb

```
  def create
    @order = Order.new(order_params)
    @order.add_line_items_from_cart(@cart)

    respond_to do |format|
      if @order.save
        Cart.destroy(session[:cart_id])
        session[:cart_id] = nil
➤       OrderMailer.received(@order).deliver_later
        format.html { redirect_to store_index_url, notice:
          'Thank you for your order.' }
        format.json { render :show, status: :created,
          location: @order }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @order.errors,
          status: :unprocessable_entity }
      end
    end
  end
```

And we need to update shipped() as we did for received():

rails7/depot_qa/app/mailers/order_mailer.rb

```
def shipped(order)
  @order = order

  mail to: order.email, subject: 'Pragmatic Store Order Shipped'
end
```

Now we have enough of the basics in place that you can place an order and have a plain email sent to yourself, assuming you didn't disable the sending of email in development mode. Let's spice up the email with a bit of formatting.

Delivering Multiple Content Types

Some people prefer to receive email in plain-text format, while others like the look of an HTML email. Rails supports this directly, allowing you to send email messages that contain alternative content formats, allowing users (or their email clients) to decide which they'd prefer to view.

3. http://api.rubyonrails.org/classes/ActionMailer/MessageDelivery.html#method-i-deliver_now

In the preceding section, we created a plain-text email. The view file for our received action was called received.text.erb. This is the standard Rails naming convention. We can also create HTML-formatted emails.

Let's try this with the order-shipped notification. We don't need to modify any code—we simply need to create a new template:

rails7/depot_qa/app/views/order_mailer/shipped.html.erb

```
<h3>Pragmatic Order Shipped</h3>
<p>
  This is just to let you know that we've shipped your recent order:
</p>

<table>
  <tr><th colspan="2">Qty</th><th>Description</th></tr>
<%= render @order.line_items -%>
</table>
```

We don't need to modify the partial, because the existing one will do just fine:

rails7/depot_qa/app/views/line_items/_line_item.html.erb

```
<% if line_item == @current_item %>
<tr class="line-item-highlight">
<% else %>
<tr>
<% end %>
  <td class="text-right"><%= line_item.quantity %></td>
  <td>&times;</td>
  <td class="pr-2">
    <%= line_item.product.title %>
  </td>
  <td class="text-right font-bold">
    <%= number_to_currency(line_item.total_price) %>
  </td>
</tr>
```

But for email templates, Rails provides a bit more naming magic. If you create multiple templates with the same name but with different content types embedded in their filenames, Rails will send all of them in one email, arranging the content so that the email client can distinguish each.

This means you'll want to either update or delete the plain-text template that Rails provided for the shipped notifier.

Testing Email

When we used the generate script to create our order mailer, it automatically constructed a corresponding order_test.rb file in the application's test/mailers directory. It's pretty straightforward; it simply calls each action and verifies

selected portions of the email produced. Because we've tailored the email, let's update the test case to match:

rails7/depot_qa/test/mailers/order_mailer_test.rb

```
require "test_helper"

class OrderMailerTest < ActionMailer::TestCase
  test "received" do
    mail = OrderMailer.received(orders(:one))
    assert_equal "Pragmatic Store Order Confirmation", mail.subject
    assert_equal ["dave@example.org"], mail.to
    assert_equal ["depot@example.com"], mail.from
    assert_match /1 x Programming Ruby 1.9/, mail.body.encoded
  end

  test "shipped" do
    mail = OrderMailer.shipped(orders(:one))
    assert_equal "Pragmatic Store Order Shipped", mail.subject
    assert_equal ["dave@example.org"], mail.to
    assert_equal ["depot@example.com"], mail.from
    assert_match %r(
      <td[^>]*>1<\/td>\s*
      <td>&times;<\/td>\s*
      <td[^>]*>\s*Programming\sRuby\s1.9\s*</td>
    )x, mail.body.to_s
  end

end
```

The test method instructs the mail class to create (but not to send) an email, and we use assertions to verify that the dynamic content is what we expect. Note the use of assert_match() to validate just part of the body content. Your results may differ depending on how you tailored the default :from line in your OrderMailer.

Note that it's also possible to have your Rails application receive emails. We'll cover that in Chapter 16, Task K: Receive Emails and Respond with Rich Text, on page 247.

Now that we've implemented our mailer and tested it, let's move on to that pesky slow payment processor. To deal with that, we'll put our API calls into a job that can be run in the background so the user doesn't have to wait.

Iteration H2: Connecting to a Slow Payment Processor with Active Job

The code inside the controllers is relatively fast and returns a response to the user quickly. This means we can reliably give users feedback by checking and validating their orders and the users won't have to wait too long for a response.

The more we add to the controller, the slower it'll become. Slow controllers create several problems. First, the user must wait a long time for a response even though the processing that's going on might not be relevant to the user experience. In the previous section, we set up sending email. The user certainly needs to get that email but doesn't need to wait for Rails to format and send it just to show a confirmation in the browser.

The second problem caused by slow code is *timeouts*. A timeout is when Rails, a web server, or a browser decides that a request has taken too long and terminates it. This is jarring to the user *and* to the code because it means the code is interrupted at a potentially odd time. What if we've recorded the order but haven't sent the email? The customer won't get a notification.

In the common case of sending email, Rails handles sending it in the background. We use `deliver_later()` to trigger sending an email, and Rails executes that code in the background. This means that users don't have to wait for email to be sent before we render a response. This is a great hidden benefit to Rails' integrated approach to building a web app.

Rails achieves this using Active Job, which is a generic framework for running code in the background. We'll use this framework to connect to the slow payment processor.

To make this change, you'll implement the integration with the payment processor as a method inside `Order`, then have the controller use Active Job to execute that method in a background job. Because the end result will be somewhat complex, you'll write a system test to ensure everything is working together.

Moving Logic into the Model

It's way outside the scope of this book to integrate with an actual payment processor, so we've cooked up a fake one named Pago, along with an implementation, which we'll see in a bit. First, this is the API it provides and a sketch of how you can use it:

```
payment_result = Pago.make_payment(
  order_id: order.id,
  payment_method: :check,
  payment_details: { routing: xxx, account: yyy }
)
```

The fake implementation does some basic validations of the parameters, prints out the payment details it received, pauses for a few seconds, and returns a structure that responds to `succeeded?()`.

rails7/depot_qb/lib/pago.rb
```
require 'ostruct'
class Pago
  def self.make_payment(order_id:,
                        payment_method:,
                        payment_details:)

    case payment_method
    when :check
      Rails.logger.info "Processing check: " +
        payment_details.fetch(:routing).to_s + "/" +
        payment_details.fetch(:account).to_s
    when :credit_card
      Rails.logger.info "Processing credit_card: " +
        payment_details.fetch(:cc_num).to_s + "/" +
        payment_details.fetch(:expiration_month).to_s + "/" +
        payment_details.fetch(:expiration_year).to_s
    when :po
      Rails.logger.info "Processing purchase order: " +
        payment_details.fetch(:po_num).to_s
    else
      raise "Unknown payment_method #{payment_method}"
    end
    sleep 3 unless Rails.env.test?
    Rails.logger.info "Done Processing Payment"
    OpenStruct.new(succeeded?: true)
  end
end
```

If you aren't familiar with OpenStruct, it's part of Ruby's standard library and provides a quick-and-dirty way to make an object that responds to the methods given to its constructor.[4] In this case, we can call succeeded?() on the return value from make_payment(). OpenStruct is handy for creating realistic objects from prototype or faked-out code like Pago.

With the payment API in hand, you need logic to adapt the payment details that you added in Defining Additional Fields, on page 179, to Pago's API. You'll also move the call to OrderMailer into this method, because you don't want to send the email if there was a problem collecting payment.

In a Rails app, when a bit of logic becomes more complex than a line or two of code, you want to move that out of the controller and into a model. You'll create a new method in Order called charge!() that will handle all this logic.

To prepare for this, we first define a pay_type_params() method in the controller that will capture the parameters to be passed to the model. We put this new method in the bottom of the controller, in the private section:

4. https://ruby-doc.org/stdlib-2.4.1/libdoc/ostruct/rdoc/OpenStruct.html

rails7/depot_qb/app/controllers/orders_controller.rb

```
def pay_type_params
  if order_params[:pay_type] == "Credit card"
    params.require(:order).permit(:credit_card_number, :expiration_date)
  elsif order_params[:pay_type] == "Check"
    params.require(:order).permit(:routing_number, :account_number)
  elsif order_params[:pay_type] == "Purchase order"
    params.require(:order).permit(:po_number)
  else
    {}
  end
end
```

The method will be somewhat long and has to do three things. First, it must adapt the `pay_type_params` that you just created to the parameters that `Pago` requires. Second, it should make the call to Pago to collect payment. Finally, it must check to see if the payment succeeded and, if so, send the confirmation email. Here's what the method looks like:

rails7/depot_qb/app/models/order.rb

```
require 'active_model/serializers/xml'
require 'pago'

class Order < ApplicationRecord
  include ActiveModel::Serializers::Xml
  enum pay_type: {
    "Check"          => 0,
    "Credit card"    => 1,
    "Purchase order" => 2
  }
  has_many :line_items, dependent: :destroy
  # ...
  validates :name, :address, :email, presence: true
  validates :pay_type, inclusion: pay_types.keys
  def add_line_items_from_cart(cart)
    cart.line_items.each do |item|
      item.cart_id = nil
      line_items << item
    end
  end

  def charge!(pay_type_params)
    payment_details = {}
    payment_method = nil

    case pay_type
    when "Check"
      payment_method = :check
      payment_details[:routing] = pay_type_params[:routing_number]
      payment_details[:account] = pay_type_params[:account_number]
    when "Credit card"
```

```
        payment_method = :credit_card
        month,year = pay_type_params[:expiration_date].split(//)
        payment_details[:cc_num] = pay_type_params[:credit_card_number]
        payment_details[:expiration_month] = month
        payment_details[:expiration_year] = year
      when "Purchase order"
        payment_method = :po
        payment_details[:po_num] = pay_type_params[:po_number]
      end

      payment_result = Pago.make_payment(
        order_id: id,
        payment_method: payment_method,
        payment_details: payment_details
      )

      if payment_result.succeeded?
        OrderMailer.received(self).deliver_later
      else
        raise payment_result.error
      end
    end
end
```

If you weren't concerned with how slow Pago's API is, you'd change the code in the create() method of OrdersController to call charge!():

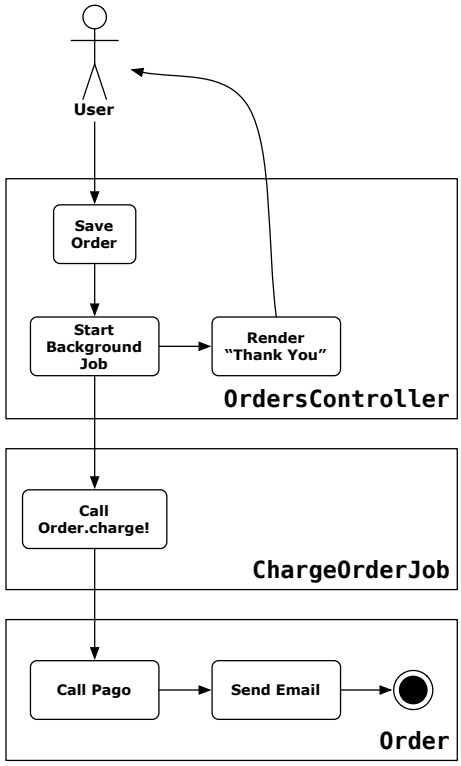
```
if @order.save
  Cart.destroy(session[:cart_id])
  session[:cart_id] = nil
  @order.charge!(pay_type_params) # do not do this
  format.html { redirect_to store_index_url, notice:
    'Thank you for your order.' }
```

Since you already know the call to Pago will be slow, you want it to happen in a background job so that users can see the confirmation message in their browser immediately without having to wait for the charge to actually happen. To do this, you must create an Active Job class, implement that class to call charge!(), and then add code to the controller to execute this job. The flow looks like the figure shown on page 201.

Creating an Active Job Class

Rails provides a generator to create a shell of a job class for us. Create the job using it like so:

```
> bin/rails generate job charge_order
    invoke  test_unit
    create    test/jobs/charge_order_job_test.rb
    create  app/jobs/charge_order_job.rb
```



The argument charge_order tells Rails that the job's class name should be ChargeOrderJob.

You've implemented the logic in the charge!() method of Order, so what goes in the newly created ChargeOrderJob? The purpose of job classes like ChargeOrderJob is to act as a glue between the controller——which wants to run some logic later——and the actual logic in the models.

Here's the code that implements this:

rails7/depot_qb/app/jobs/charge_order_job.rb

```
class ChargeOrderJob < ApplicationJob
  queue_as :default

  def perform(order,pay_type_params)

    order.charge!(pay_type_params)

  end
end
```

Next, you need to fire this job in the background from the controller.

Queuing a Background Job

Because background jobs run in parallel to the code in the controller, the code you write to initiate the background job isn't the same as calling a method. When you call a method, you expect that method's code to be executed while you wait. Background jobs are different. They often go to a queue, where they wait to be executed outside the controller. Thus, when we talk about executing code in a background job, we often use the phrase "queue the job."

To queue a job using Active Job, use the method perform_later() on the job class and pass it the arguments you want to be given to the perform() method you implemented above. Here's where to do that in the controller (note that this replaces the call to OrderMailer, since that's now part of the charge!() method):

rails7/depot_qb/app/controllers/orders_controller.rb

```
  def create
    @order = Order.new(order_params)
    @order.add_line_items_from_cart(@cart)

    respond_to do |format|
      if @order.save
        Cart.destroy(session[:cart_id])
        session[:cart_id] = nil
➤       ChargeOrderJob.perform_later(@order,pay_type_params.to_h)
        format.html { redirect_to store_index_url, notice:
          'Thank you for your order.' }
        format.json { render :show, status: :created,
          location: @order }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @order.errors,
          status: :unprocessable_entity }
      end
    end
  end
```

With this in place, you can now add an item to the cart, check out, and see everything working just as we did before, with the addition of seeing the calls to Pago. If you look at the Rails log when you check out, you should see some logging, like so (formatted to fit the page):

```
[ActiveJob] Enqueued ChargeOrderJob
  (Job ID: 79da671e-865c-4d51-a1ff-400208c6dbd1)
    to Async(default) with arguments:
      #<GlobalID:0x007fa294a43ce0 @uri=#<URI::GID gid://depot/Order/9>>,
      {"routing_number"=>"23412341234", "account_number"=>"345356345"}
[ActiveJob] [ChargeOrderJob] [79da671e-865c-4d51-a1ff-400208c6dbd1]
    Performing ChargeOrderJob
      (Job ID: 79da671e-865c-4d51-a1ff-400208c6dbd1) from
```

```
    Async(default) with arguments:
    #<GlobalID:0x007fa294a01570 @uri=#<URI::GID gid://depot/Order/9>>,
    {"routing_number"=>"23412341234", "account_number"=>"345356345"}
[ActiveJob] [ChargeOrderJob] [79da671e-865c-4d51-a1ff-400208c6dbd1]
    Processing check: 23412341234/345356345
```

This shows the guts of how Active Job works and is useful for debugging if things aren't working right.

Speaking of debugging and possible failures, this interaction really should have a test.

System Testing the Checkout Flow

In Iteration G3: Testing Our JavaScript Functionality, on page 183, you wrote a system test that uses a real browser to simulate user interaction. To test the entire flow of checking out, communicating with the payment processor, and sending an email, you'll add a second test.

To test the full, end-to-end workflow, including execution of Active Jobs, you want to do the following:

1. Add a book to the cart.
2. Fill in the checkout form completely (including selecting a pay type).
3. Submit the order.
4. Process all background jobs.
5. Check that the order was created properly.
6. Check that email was sent.

You should already be familiar with how to write most parts of this test. Processing background jobs and checking mail, however, are new. Rails provides helpers for us, so the test will be short and readable when you're done. One of those helpers is available by mixing in the ActiveJob::TestHelper module:

rails7/depot_qb/test/system/orders_test.rb

```
class OrdersTest < ApplicationSystemTestCase
  include ActiveJob::TestHelper
```

This provides the method perform_enqueued_jobs(), which you'll use in your test:

rails7/depot_qb/test/system/orders_test.rb

```
test "check order and delivery" do
  LineItem.delete_all
  Order.delete_all

  visit store_index_url

  click_on 'Add to Cart', match: :first

  click_on 'Checkout'
```

```
  fill_in 'Name', with: 'Dave Thomas'
  fill_in 'Address', with: '123 Main Street'
  fill_in 'Email', with: 'dave@example.com'

  select 'Check', from: 'Pay type'
  fill_in "Routing number", with: "123456"
  fill_in "Account number", with: "987654"

  click_button "Place Order"
  assert_text 'Thank you for your order'

  perform_enqueued_jobs
  perform_enqueued_jobs
  assert_performed_jobs 2

  orders = Order.all
  assert_equal 1, orders.size

  order = orders.first
  assert_equal "Dave Thomas",      order.name
  assert_equal "123 Main Street",  order.address
  assert_equal "dave@example.com", order.email
  assert_equal "Check",            order.pay_type
  assert_equal 1, order.line_items.size

  mail = ActionMailer::Base.deliveries.last
  assert_equal ["dave@example.com"],                mail.to
  assert_equal 'Sam Ruby <depot@example.com>',      mail[:from].value
  assert_equal "Pragmatic Store Order Confirmation", mail.subject
end
```

This test reads almost like English. Since you now need to submit the form and assert that an order was created, you start by clearing out any orders in the test database that might be hanging around from previous test runs.

Next, you add an item to the cart, check out and fill in the pay type details, place your order, and verify that you get a *Thank you* response.

Since this test is about the user's experience end-to-end, you don't need to look at the jobs that have been queued—instead we need to make sure they are executed. It's sufficient to assert the *results* of those jobs having been executed. To that end, the method perform_enqueued_jobs() will perform any jobs that get enqueued inside the block of code given to it.Since our ChangeOrderJob enqueues a mail job, clearing the queue once isn't enough, so we clear it twice. After this, we verify that exactly two jobs were executed.

Next, check that an order was created in the way you expect by locating the created order and asserting that the values provided in the checkout form were properly saved.

Joe asks:

How Are Background Jobs Run in Development or Production?

When running the application locally, the background jobs are executed and emails are sent by Rails. By default, Rails uses an in-memory queue to manage the jobs. This is fine for development, but it could be a problem in production. If your app were to crash before all background jobs were processed or before emails were sent, those jobs would be lost and unrecoverable.

In production, you'd need to use a different *back end*, as detailed in the Active Job Rails Guide.[a] Sidekiq is a popular open source back end that works great.[b] Setting it up is a bit tricky since you must have access to a Redis database to store the waiting jobs.[c] If you're using Postgres for your Active Records, Queue Classic is another option for a back end that doesn't require Redis—it uses your existing Postgres database.[d]

a. http://guides.rubyonrails.org/active_job_basics.html#job-execution
b. http://sidekiq.org/
c. https://redis.io/
d. https://github.com/QueueClassic/queue_classic/tree/3-1-stable

Lastly, you need to check that the mail was sent. In the test environment, Rails doesn't actually deliver mail but instead saves it in an array available via ActionMailer::Base.deliveries(). The objects in there respond to various methods that allow you to examine the email:

If you run this test via bin/rails test test/system/orders_test.rb, it should pass. You've now tested a complex workflow using the browser, background jobs, and email.

What We Just Did

Without much code and with just a few templates, we've managed to pull off the following:

- We configured our development, test, and production environments for our Rails application to enable the sending of outbound emails.
- We created and tailored a mailer that can send confirmation emails in both plain-text and HTML formats to people who order our products.
- We used Active Job to execute slow-running code in the background so the user doesn't have to wait.

- We enhanced a system test to cover the entire end-to-end workflow, including verifying that the background job executed and the email was sent.

Playtime

Here's some stuff to try on your own:

- Add a `ship_date` column to the orders table, and send a notification when this value is updated by the `OrdersController`.
- Update the application to send an email to the system administrator—namely, yourself—when an application failure occurs, such as the one we handled in Iteration E2: Handling Errors, on page 132.
- Modify `Pago` to sometimes return a failure (`OpenStruct.new(succeeded?: false)`), and handle that by sending a different email with the details of the failure.
- Add system tests for all of the above.