```
    assert_match FILL_IN, response.body
  end
end
```

13.4 Micropost images

Now that we've added support for all relevant micropost actions, in this section we'll make it possible for microposts to include images as well as text. We'll start with a basic version good enough for development use, and then add a series of enhancements to make image upload production-ready.

Adding image upload involves two main visible elements: a form field for uploading an image and the micropost images themselves. A mockup of the resulting "Upload image" button and micropost photo appears in Figure 13.23.[16]

13.4.1 Basic image upload

The most convenient way to upload files in Rails is to use a built-in feature called Active Storage.[17] Active Storage makes it easy to handle an uploaded image and associate it with a model of our choice (e.g., the Micropost model). Although we'll be using it only for uploading images, Active Storage is actually quite general, and can handle plain text and multiple kinds of binary files (such as PDF documents or recorded audio).

As described in the Active Storage documentation, adding Active Storage to our application is as easy as running a single command:

```
$ rails active_storage:install
```

This command generates a database migration that creates a data model for storing attached files. You're welcome to take a look at it, but this is an excellent

[16] Image retrieved from https://www.flickr.com/photos/grungepunk/14026922186 on 2014-09-19. Copyright © 2014 by Jussie D. Brito and used unaltered under the terms of the Creative Commons Attribution-ShareAlike 2.0 Generic license.

[17] Active Storage was added in Rails 5.2.

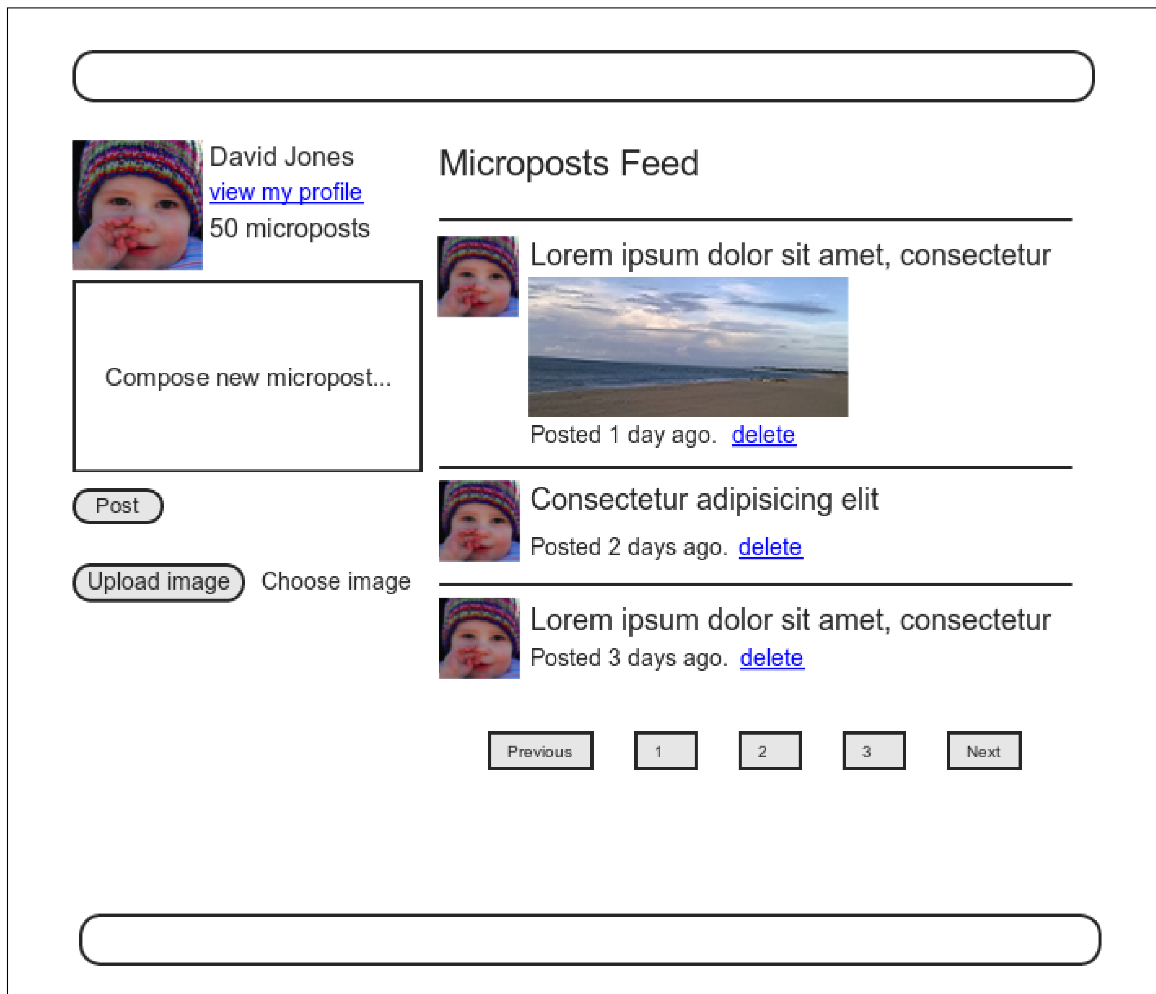

Figure 13.23: A mockup of micropost image upload (with an uploaded image).

example of applying technical sophistication to know which details matter and which don't. In this case, what matters is the API for interacting with Active Storage, which we'll start covering in a moment; for our purposes, the implementation details are safe to ignore. All we need to do to set it up is run the migration:

```
$ rails db:migrate
```

The first part of the Active Storage API that we need is the **`has_one_attached`** method, which allows us to associate an uploaded file with a given model. In our case, we'll call it **`image`** and associate it with the Micropost model, as shown in Listing 13.59.

Listing 13.59: Adding an image to the Micropost model.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to       :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

We'll adopt a design of one image per micropost for our application, but Active Storage also offers a second option, **`has_many_attached`**, which allows for the attachment of multiple files to a single Active Record object.

To include image upload on the Home page as in Figure 13.23, we need to include a **`file_field`** tag in the micropost form, as shown in Listing 13.60 and Figure 13.24.

Listing 13.60: Adding image upload to the micropost create form.
app/views/shared/_micropost_form.html.erb

```
<%= form_with(model: @micropost, local: true) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
```

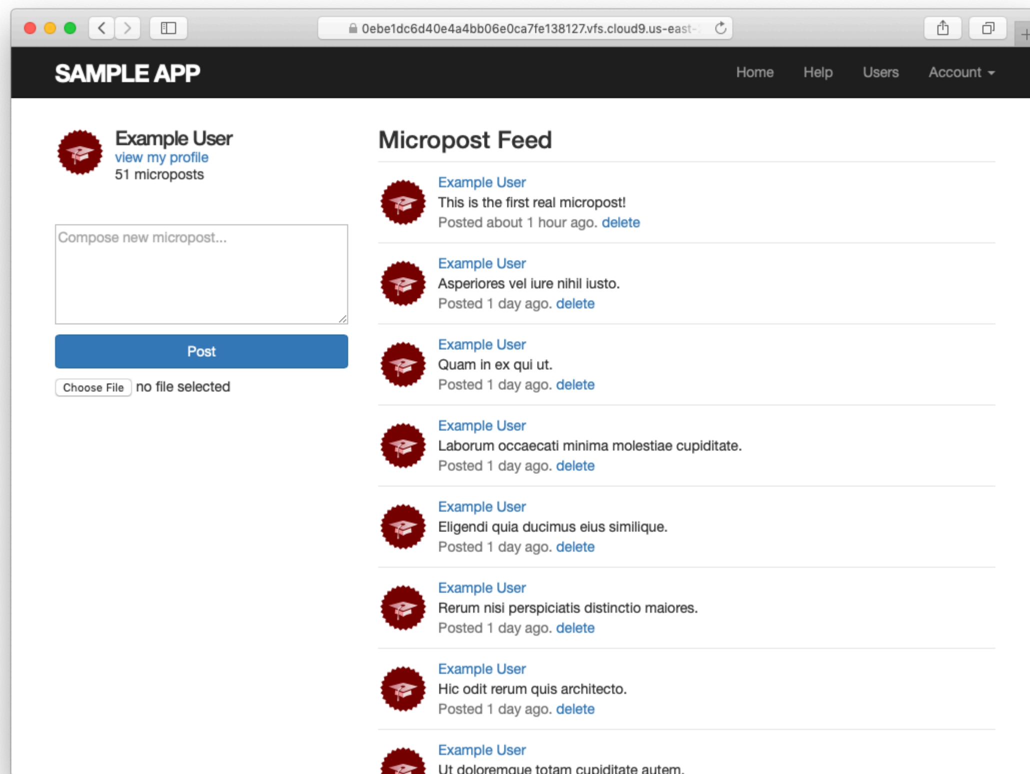

Figure 13.24: Adding an image upload field.

```
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
  <span class="image">
    <%= f.file_field :image %>
  </span>
<% end %>
```

Finally, we need to update the Microposts controller to add the image to the newly created micropost object. We can do this using the **`attach`** method provided by the Active Storage API, which attaches the uploaded image to the **`@micropost`** object in the Microposts controller's **`create`** action. To allow

the upload to go through, we also need to update **`micropost_params`** method to add **`:image`** to the list of attributes permitted to be modified through the web. The result appears in Listing 13.61.

Listing 13.61: Adding **`image`** to the list of permitted attributes.
app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]
  before_action :correct_user,   only: :destroy

  def create
    @micropost = current_user.microposts.build(micropost_params)
    @micropost.image.attach(params[:micropost][:image])
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_url
    else
      @feed_items = current_user.feed.paginate(page: params[:page])
      render 'static_pages/home'
    end
  end

  def destroy
    @micropost.destroy
    flash[:success] = "Micropost deleted"
    redirect_to request.referrer || root_url
  end

  private

    def micropost_params
      params.require(:micropost).permit(:content, :image)
    end

    def correct_user
      @micropost = current_user.microposts.find_by(id: params[:id])
      redirect_to root_url if @micropost.nil?
    end
end
```

Once the image has been uploaded, we can render the associated **`micropost.image`** using the **`image_tag`** helper in the micropost partial, as shown in Listing 13.62. Notice the use of the **`attached?`** boolean method to prevent displaying an image tag when there isn't an image.

Listing 13.62: Adding image display to microposts.
app/views/microposts/_micropost.html.erb

```
<li id="micropost-<%= micropost.id %>">
  <%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
  <span class="user"><%= link_to micropost.user.name, micropost.user %></span>
  <span class="content">
    <%= micropost.content %>
    <%= image_tag micropost.image if micropost.image.attached? %>
  </span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
    <% if current_user?(micropost.user) %>
      <%= link_to "delete", micropost, method: :delete,
                                       data: { confirm: "You sure?" } %>
    <% end %>
  </span>
</li>
```

The result of making a micropost with an image appears in Figure 13.25. I'm always amazed when things like this actually work, but there's the proof! (It's also a good idea to write at least a basic automated test for image upload, which is left as an exercise (Section 13.4.1).)

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Upload a micropost with attached image. Does the result look too big? (If so, don't worry; we'll fix it in Section 13.4.3.)

2. Following the template in Listing 13.64, write a test of the image uploader in Section 13.4. As preparation, you should add an image to the fixtures directory using Listing 13.63. The additional assertions in Listing 13.64 check both for a file upload field on the Home page and for a valid image attribute on the micropost resulting from valid submission. Note the use of the special **`fixture_file_upload`** method for uploading files as

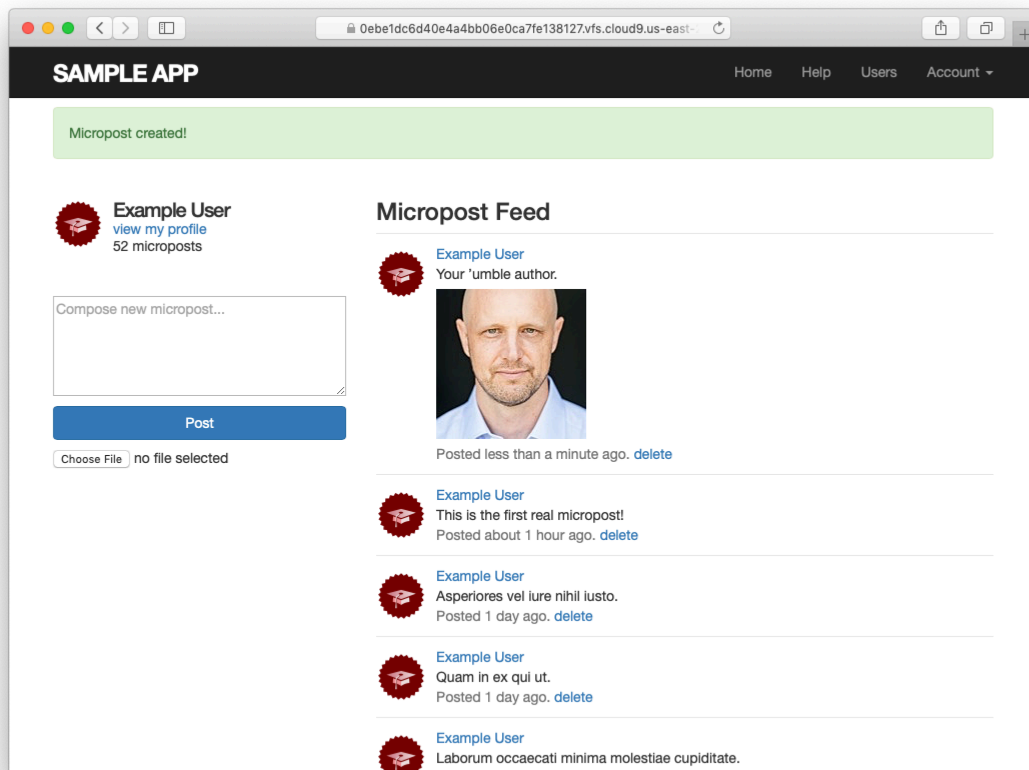

Figure 13.25: The result of submitting a micropost with an image.

fixtures inside tests.[18] *Hint*: To check for a valid **`image`** attribute, use the **`assigns`** method mentioned in Section 11.3.3 to access the micropost in the **`create`** action after valid submission.

Listing 13.63: Downloading a fixture image for use in the tests.

```
$ curl -o test/fixtures/kitten.jpg -OL https://cdn.learnenough.com/kitten.jpg
```

Listing 13.64: A template for testing image upload.
`test/integration/microposts_interface_test.rb`

```
require 'test_helper'

class MicropostInterfaceTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "micropost interface" do
    log_in_as(@user)
    get root_path
    assert_select 'div.pagination'
    assert_select 'input[type=FILL_IN]'
    # Invalid submission
    assert_no_difference 'Micropost.count' do
      post microposts_path, params: { micropost: { content: "" } }
    end
    assert_select 'div#error_explanation'
    assert_select 'a[href=?]', '/?page=2'  # Correct pagination link
    # Valid submission
    content = "This micropost really ties the room together"
    image = fixture_file_upload('test/fixtures/kitten.jpg', 'image/jpeg')
    assert_difference 'Micropost.count', 1 do
      post microposts_path, params: { micropost:
                                      { content: content, image: image } }
    end
    assert FILL_IN.image.attached?
    follow_redirect!
    assert_match content, response.body
    # Delete a post.
    assert_select 'a', text: 'delete'
    first_micropost = @user.microposts.paginate(page: 1).first
```

[18] Windows users should add a **`:binary`** parameter: **`fixture_file_upload(file, type, :binary)`**.

```
    assert_difference 'Micropost.count', -1 do
      delete micropost_path(first_micropost)
    end
    # Visit a different user.
    get user_path(users(:archer))
    assert_select 'a', { text: 'delete', count: 0 }
  end
  .
  .
  .
end
```

13.4.2 Image validation

The image upload code in Section 13.4.1 is a good start, but it has significant limitations. Among other things, it doesn't enforce any constraints on the uploaded file, which can cause problems if users try to upload large files or invalid file types. To remedy this defect, we'll add validations for the image size and format.

As of this writing, Active Storage (somewhat surprisingly) doesn't offer native support for things like format and size validations, but as is so often the case there is a gem that adds it for us (Listing 13.65).

Listing 13.65: Adding a gem for Active Storage validations.
Gemfile

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                      '6.0.1'
gem 'active_storage_validations', '0.8.2'
gem 'bcrypt',                     '3.1.13'
.
.
.
```

Then **`bundle install`**:

```
$ bundle install
```

Following the gem documentation, we see that we can validate the image format by examining the **`content_type`** as follows:

```
content_type: { in: %w[image/jpeg image/gif image/png],
                message: "must be a valid image format" }
```

This checks that the MIME type of the image corresponds to a supported image format. (Recall the **`%w[]`** array-building syntax from Section 6.2.4.)

Similarly, we can validate the file size like this:

```
size: { less_than: 5.megabytes,
        message:   "should be less than 5MB" }
```

This sets a limit of 5 megabytes using a syntax we saw before in the context of time helpers (Box 9.1).

Adding these validations to the Micropost model gives the code in Listing 13.66.

Listing 13.66: Adding validations to images.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to       :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
  validates :image,   content_type: { in: %w[image/jpeg image/gif image/png],
                                      message: "must be a valid image format" },
                      size:         { less_than: 5.megabytes,
                                      message:   "should be less than 5MB" }
end
```

The result of trying to upload a large, invalid image then appears as in Figure 13.26. (You may have to restart the Rails server first.)

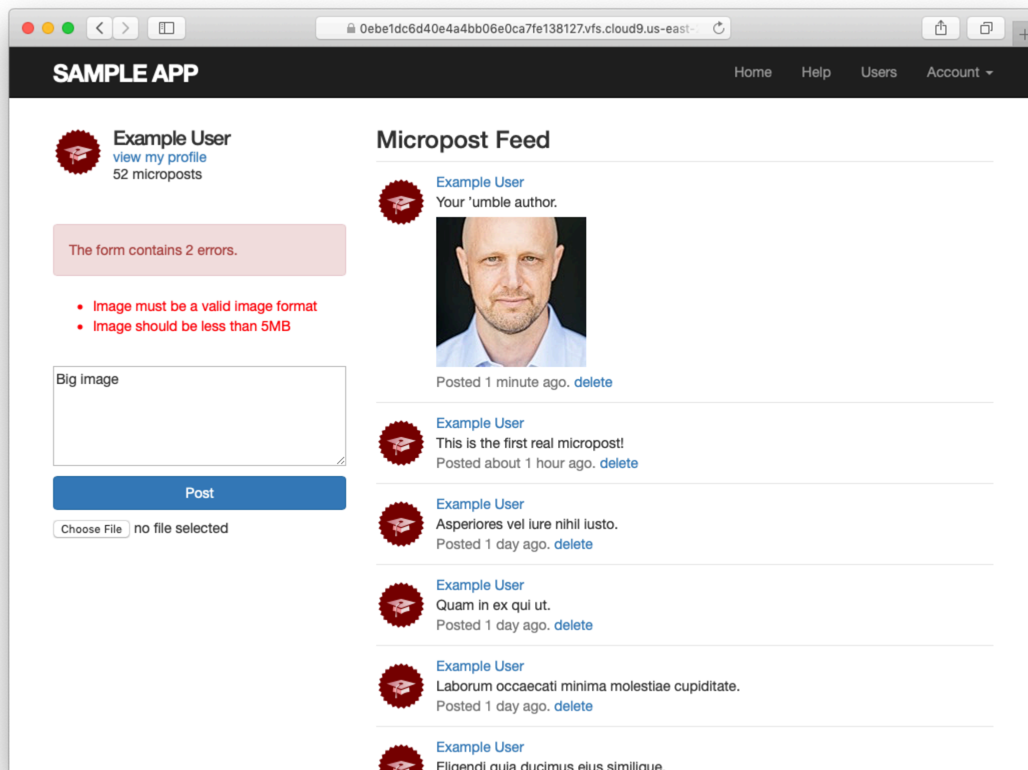

Figure 13.26: Trying to upload a large, invalid image.

To go along with the validations in Listing 13.66, we'll add client-side (in-browser) checks on the uploaded image size and format. We'll start by including a little JavaScript (or, more specifically, jQuery) to issue an alert if a user tries to upload an image that's too big (which prevents accidental time-consuming uploads and lightens the load on the server). The result appears in Listing 13.67.[19]

Listing 13.67: Checking the file size with jQuery.
app/views/shared/_micropost_form.html.erb

```
<%= form_with(model: @micropost, local: true) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
  <span class="image">
    <%= f.file_field :image %>
  </span>
<% end %>

<script type="text/javascript">
  $("#micropost_image").bind("change", function() {
    var size_in_megabytes = this.files[0].size/1024/1024;
    if (size_in_megabytes > 5) {
      alert("Maximum file size is 5MB. Please choose a smaller file.");
      $("#micropost_image").val("");
    }
  });
</script>
```

Although JavaScript isn't the focus of this book, you might be able to figure out that Listing 13.67 monitors the page element containing the CSS id **`micropost_image`** (as indicated by the hash mark `#`), which is the id of the micropost form in Listing 13.60. (The way to figure this out is to Ctrl-click and use your browser's web inspector.) When the element with that CSS id changes, the jQuery function fires and issues the **`alert`** method if the file is too big, as seen in Figure 13.27.[20]

[19] More advanced users of JavaScript would probably put the size check in its own function, but since this isn't a JavaScript tutorial the code in Listing 13.67 is fine for our purposes.

[20] To learn how to do things like this, you can do what I did: Google for things like "javascript maximum file size" until you find something on Stack Overflow.

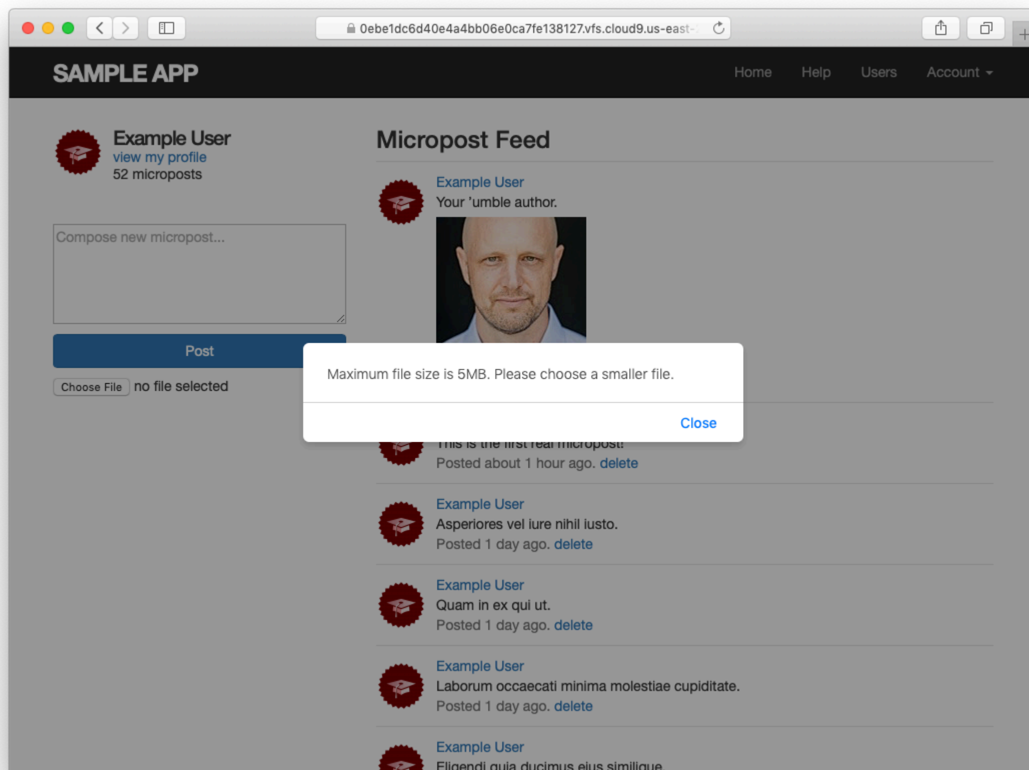

Figure 13.27: A JavaScript alert for a large file.

Finally, by using the **accept** parameter in the **file_field** input tag, we can specify that only valid formats should be allowed (Listing 13.68).

Listing 13.68: Allowing only valid image formats.
app/views/shared/_micropost_form.html.erb

```
<%= form_with(model: @micropost, local: true) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
  <span class="image">
    <%= f.file_field :image, accept: "image/jpeg,image/gif,image/png" %>
  </span>
<% end %>

<script type="text/javascript">
  $("#micropost_image").bind("change", function() {
    var size_in_megabytes = this.files[0].size/1024/1024;
    if (size_in_megabytes > 5) {
      alert("Maximum file size is 5MB. Please choose a smaller file.");
      $("#micropost_image").val("");
    }
  });
</script>
```

Listing 13.68 arranges to allow only valid image types to be selected in the first place, graying out any other file types (Figure 13.28).

Preventing invalid images from being uploaded in a browser is a nice touch, but it's important to understand that this sort of code can only make it *difficult* to upload an invalid format or large file; a user determined to upload an invalid file can always issue a direct `POST` request using, e.g., **curl**. It is thus essential to include server-side validations of the type shown in Listing 13.66.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. What happens if you try uploading an image bigger than 5 megabytes?

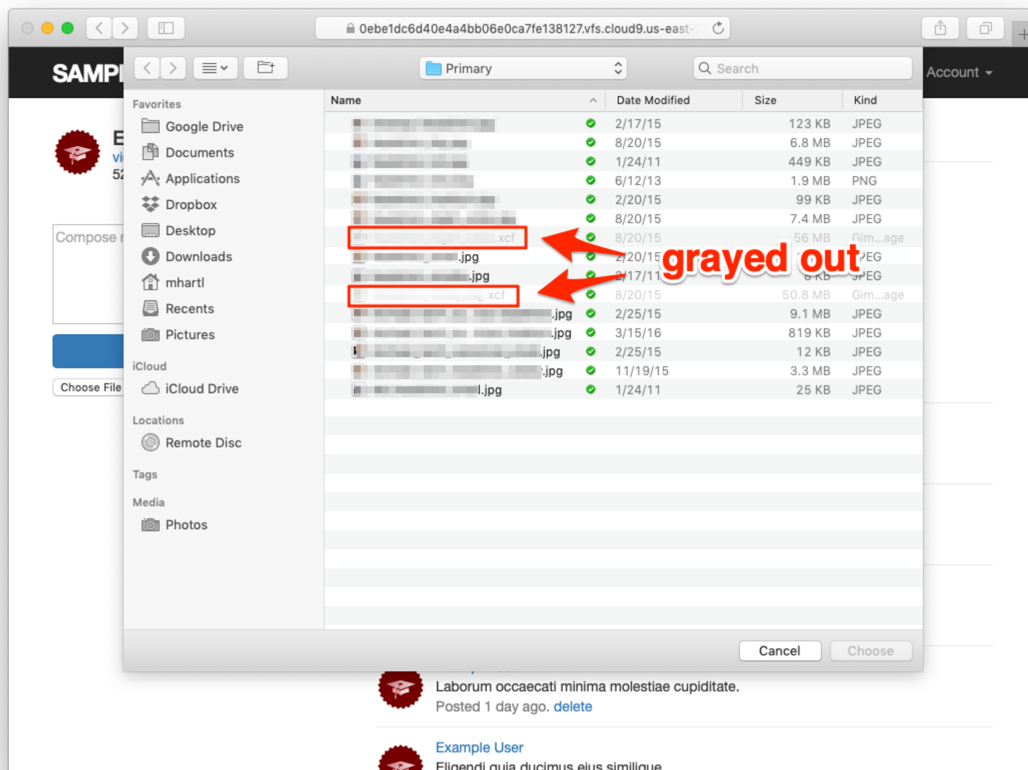

Figure 13.28: Grayed-out invalid file types.

2. What happens if you try uploading a file with an invalid extension?

13.4.3 Image resizing

The image size validations in Section 13.4.2 are a good start, but they still allow the uploading of images large enough to break our site's layout, sometimes with frightening results (Figure 13.29). Thus, while it's convenient to allow users to select fairly large images from their local disk, it's also a good idea to resize the images before displaying them.[21]

We'll be resizing images using the image manipulation program ImageMagick, which we need to install on the development environment. (As we'll see in Section 13.4.4, when using Heroku for deployment ImageMagick comes pre-installed in production.) On the cloud IDE, we can do this as follows:

```
$ sudo apt-get -y install imagemagick
```

(If you're not using the cloud IDE or an equivalent Linux system, do a Google search for "imagemagick <your platform>". On macOS, **`brew install imagemagick`** should work if you have Homebrew installed. Use your technical sophistication (Box 1.2) if you get stuck.)

Next, we need to add a couple of gems for image processing, including the aptly named `image_processing` gem and `mini_magick`, a Ruby processor for ImageMagick (Listing 13.69).

Listing 13.69: Adding gems for image processing.
`Gemfile`

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                    '6.0.1'
gem 'image_processing',         '1.9.3'
```

[21]It's possible to constrain the *display* size with CSS, but this doesn't change the image size. In particular, large images would still take a while to load. (You've probably visited websites where "small" images seemingly take forever to load. This is why.)

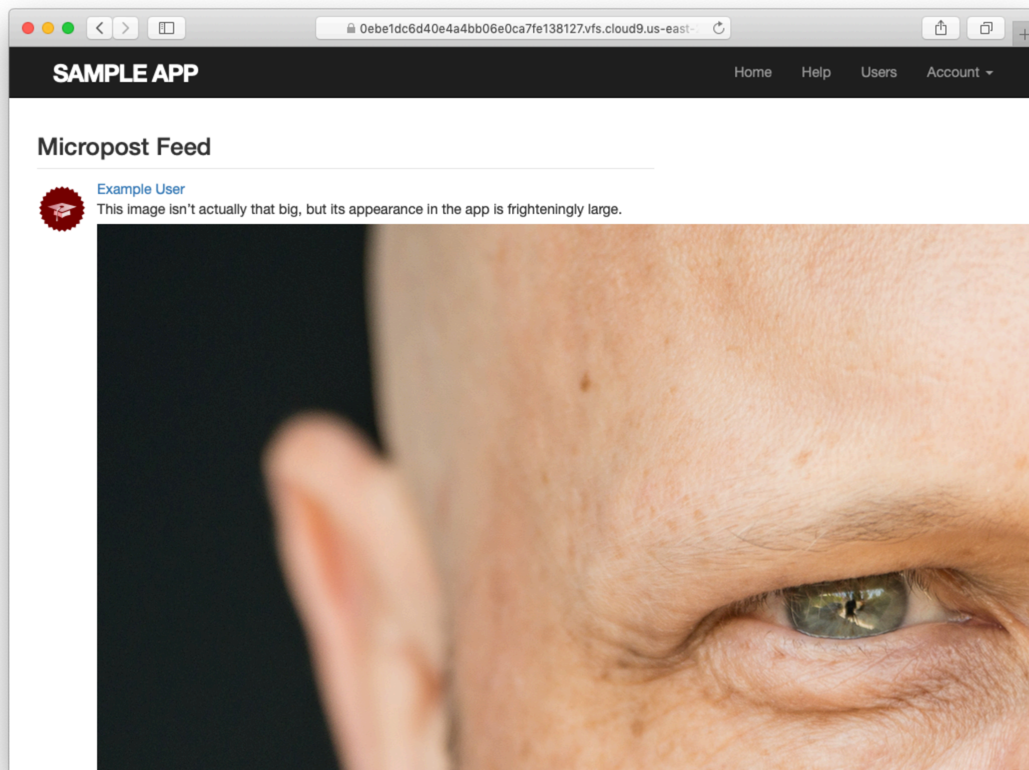

Figure 13.29: A frighteningly large uploaded image.

```
gem 'mini_magick',                '4.9.5'
gem 'active_storage_validations', '0.8.2'
.
.
.
```

Then install as usual:

```
$ bundle install
```

You will probably need to restart the Rails server as well.

With the necessary software installed, we're now ready to use the **`variant`** method supplied by Active Storage for creating transformed images. In particular, we'll use the **`resize_to_limit`** option to ensure that neither the width nor the height of the image is greater than 500 pixels, as follows:

```
image.variant(resize_to_limit: [500, 500])
```

For convenience, we'll put this code in a separate **`display_image`** method, as shown in Listing 13.70.

Listing 13.70: Adding a resized display image.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to       :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
  validates :image,   content_type: { in: %w[image/jpeg image/gif image/png],
                                      message: "must be a valid image format" },
                      size: { less_than: 5.megabytes,
                              message:     "should be less than 5MB" }

  # Returns a resized image for display.
  def display_image
    image.variant(resize_to_limit: [500, 500])
  end
end
```

Finally, we can use **`display_image`** in the micropost partial, as shown in Listing 13.71.

Listing 13.71: Using the resized **`display_image`**.
app/views/microposts/_micropost.html.erb

```
<li id="micropost-<%= micropost.id %>">
  <%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
  <span class="user"><%= link_to micropost.user.name, micropost.user %></span>
  <span class="content">
    <%= micropost.content %>
    <%= image_tag micropost.display_image if micropost.image.attached? %>
  </span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
    <% if current_user?(micropost.user) %>
      <%= link_to "delete", micropost, method: :delete,
                                       data: { confirm: "You sure?" } %>
    <% end %>
  </span>
</li>
```

The **`variant`** resizing in Listing 13.70 will happen on demand when the method is first called in Listing 13.71, and will be cached for efficiency in subsequent uses.[22] The result is a properly resized display image, as seen in Figure 13.30.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Upload a large image and confirm directly that the resizing is working. Does the resizing work even if the image isn't square?

[22]For larger sites, it's probably better to defer such processing to a background process; this method is beyond the scope of this tutorial, but investigating Active Job will get you started if you need to go this route.

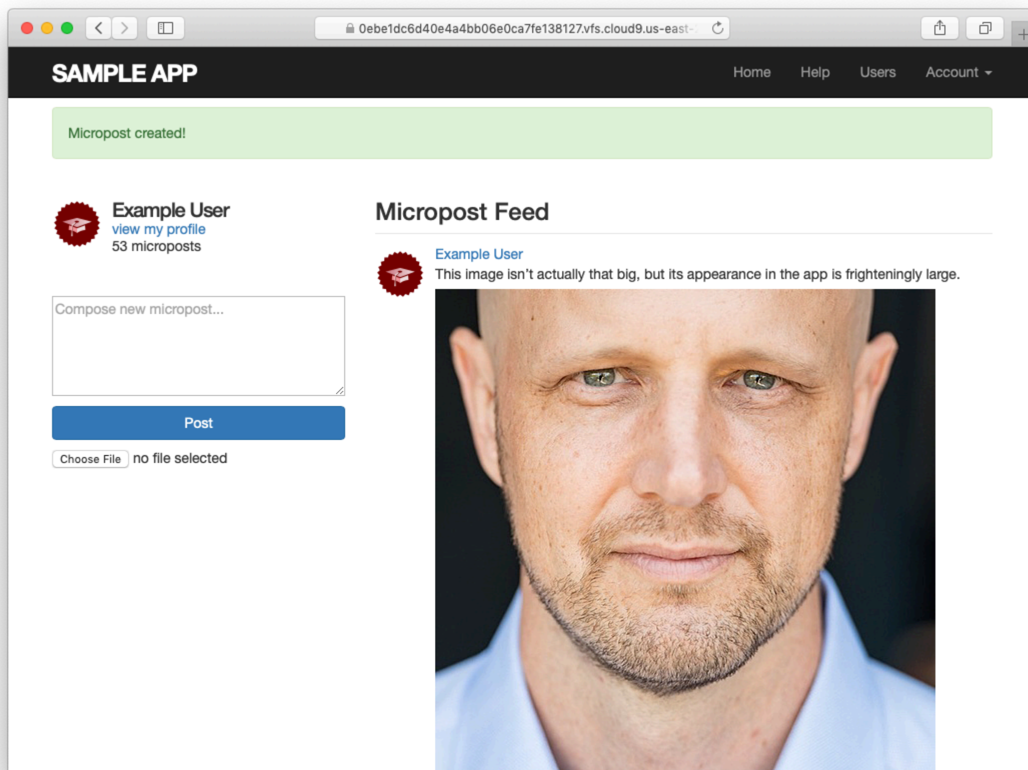

Figure 13.30: A nicely resized image.