



Thank You

Thanks for downloading this sample chapter of Keep Your Ruby on Rails App Healthy!

The premium edition of the Keep Your Ruby on Rails App Healthy of is a complete rewrite of the original course.

As well as going into more depth about the topics in the original course, it also greatly expands on the content that is covered.

This sample chapter – *Using Object Storage For Assets* – is entirely new. I hope you find it valuable, and would love to get your feedback.

Please email me (chris@plymouthsoftware.com), or drop me a message on Twitter ([@cblunt](https://twitter.com/cblunt)).

Thank you!

Use Object Storage for Assets

Media assets that are generated by your app (such as user profile photos, uploaded images, video files, etc.) should be stored in some form of object storage. Whilst there are plenty of CDN services, one of the most popular is [Amazon's S3](#) service, which is low-cost and allows infinite storage, as well as other tools such as life-cycle management.

Why use Object Storage?

Object storage breaks the dependency between your app's server (and code), and its media assets such as images uploaded by your users. In doing this, your app becomes easier to scale across multiple servers, and even move between different servers or hosting providers.

As an example, let's say your app has 20GB of user-uploaded content, stored on the same server as your app. As that storage begins to run low, you need to scale your server either vertically (add more storage, RAM, CPU, etc.), or horizontally (create a new server and load-balance requests between them).

Scaling vertically is potentially wasteful - you'll be adding unnecessary resources to your server, just to get more storage. Scaling horizontally is difficult, because all your assets are on server A, and without some work configuring network mounts and so on, server B can't access them.

What about Block Storage?

Block storage offers a middle-ground between storing your assets locally, and using an object storage provider such as S3. Block storage volumes attach to your server, and appear as mounted volumes. This means they can be accessed directly like any other part of the file system.

You could, for example, mount a block storage volume to `/assets` on your app's server, and you would be separating your app's content from its server. In theory, the block storage could be moved to another server if you needed to scale up. However, there are restrictions. For example, Digital Ocean's block storage volumes can only be attached to one server (droplet) at a time, so scaling your app horizontally is more complicated.

Using block storage would also leave serving the assets up to your app (or configuring apache/nginx to serve the assets directly). Object storage services can serve assets directly (or via CDN), removing the need for a user's browser to hit your app's server.

Object storage is (generally) pay-for-what-you-use, and limitless in size. Block storage still has a (virtual) size. You'll need to provision a block storage volume (say 10GB), and scale that up as it begins to fill. The knock-on effect of this is that you'll be paying for storage you don't use.

Whilst block storage is an option, I much prefer to delegate media and content to an Object Storage provider, such as S3.

Using Amazon S3 to Store Your App's Assets

Let's get started by setting up an S3 bucket, and configuring your app to use it when storing uploaded assets. If you don't already have an Amazon AWS account, you can [create a new account](#) and take advantage of the 12-month "free usage tier".

I'll also be using the Amazon AWS CLI tool to interact with AWS, rather than the web-based console. However, you can replicate any of these steps using the web console if you prefer.

If you don't already have it set up, install and configure the AWS CLI tool. If you're using Homebrew on macOS, the tools can be installed with:

```
$ brew install awscli
$ aws configure
...
```

For other platforms, or if you're not using Homebrew, follow [the official instructions](#). Once the AWS CLI tool is installed and configured, we can set up a new user and bucket in which to store your app's assets.

Setting up a Bucket

S3 stores files (objects) in buckets. A bucket can be thought of as a volume or disk. Each bucket has a *globally* unique name, i.e. if someone else has used a bucket name, then you cannot use it. Using a reverse domain name (e.g. **com.yourcompany.app.assets**) is usually a good way to namespace your buckets.

You should also avoid using periods (.) in your bucket name, as it can cause SSL verification errors when connecting to the bucket from your app. It's best to replace periods with a hyphen (-) in your bucket name. Using the example above, the bucket would be called **com-yourcompany-app-assets**.

Permissions and Policies

Access to buckets is configured through permissions and policies for different users and accounts. When creating a bucket, you can specify that it is accessible only to specific users, or to the public (anonymous access). You can also specify different read/write permissions. These permissions apply globally to the bucket and all sub-folders.

For more intricate access controls, you use *Policies* to specify different levels of access. For example, a policy can be used to allow only certain users to upload files into certain folders in the bucket.

ProTip 1 Amazon's permissions and policies are extremely flexible and powerful, and consequently, can become very complicated. We'll be setting up a basic security policy here, but you should read Amazon's documentation to fully understand the various options available to you.

ProTip 2 Strictly speaking, S3 has no concept of folders and subfolders - every file (object) sits at the top level. However, for simplicity, slashes (/) in the object's filename are interpreted to be folder-style prefixes. A more thorough discussion of this is available in [Amazon's documentation](#).

Create a new user for your app

Start by creating a new user in your Amazon account. This user will be used to represent your Rails app when communicating with AWS. You can then grant this user account (your app) permission to upload files into the bucket. This helps to keep your AWS account secure, and prevent "access creep" that could occur if using your root account credentials to communicate with S3.

Create a new user now using the AWS CLI tool:

```
$ aws iam create-user --user-name my-rails-app
$ aws iam create-access-key --user-name my-rails-app
{
  "AccessKey": {
    "UserName": "my-rails-app",
    "Status": "Active",
    "CreateDate": "2017-05-10T09:48:00.838Z",
    "SecretAccessKey": "abcdef1234567/abcdefghIJKLmn0pqRStUV",
    "AccessKeyId": "ABCD1234DEFG56789"
  }
}
```

This will do 2 things:

- Create a new user account in your AWS account called **my-rails-app**
- Create an access key for the **my-rails-app** user. An access key allows your app to access the AWS account without supplying a username/password, but instead providing an Access Key and Secret key.

Important You should store the `SecretAccessKey` somewhere secure (I recommend somewhere like LastPass, 1Password or KeePass). You cannot retrieve the `SecretAccessKey`, so if lost you'll need to generate a new access key for the user.

Create the Bucket

With our new user and access key set up, we can create an S3 bucket with a simple call to the Amazon CLI tool. Give your new bucket a unique name, e.g. **your-app-name-assets**.

```
$ aws s3api create-bucket --bucket your-app-name-assets --acl public-read --region eu-west-1
```

Note: I've created this bucket in the `eu-west-1` (Ireland) region. You may want to create your bucket in a region where you are based, e.g. `us-west-1`, `cn-north-1`, `ap-northeast-1`, etc. You can see the available regions at http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

Configure a Bucket Policy

The bucket was created with a `public-read` access control list (ACL), meaning that any content uploaded to the bucket will be publicly accessible. Our example app will be uploading public data (e.g. user profile images), so this is fine.

However, we'll need to provide an additional policy granting our app (or, more accurately, the **my-rails-app** user we created in the previous step) permission to upload content into the bucket. We can do that by specifying a policy for the bucket.

First, we'll need to get the *arn* for our user. This is the unique identifier for our user, and can be retrieved using the `iam get-user` command:

```
$ aws iam get-user --user my-rails-app
{
  "User": {
    "UserName": "my-rails-app",
    "Path": "/",
    "CreateDate": "2017-05-10T09:47:39Z",
    "UserId": "ABCD1234DEFG56789",
    "Arn": "arn:aws:iam::012345678:user/my-rails-app"
  }
}
```

Create a new document on your Desktop called *policy.json*, and paste the *arn* value from the previous step into the `Principal:Aws` line. Similarly, make sure the `Resource` line uses the *arn* for your bucket.

```
// ~/Desktop/policy.json
{
  "Id": "Policy1234567890",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123456778890",
      "Action": [
        "s3:DeleteObject",
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::your-app-name-assets/*",
      "Principal": [
        "AWS": ["arn:aws:iam::012345678:user/my-rails-app"]
      ]
    }
  ]
}
```

Save the policy file, and finally apply it to your bucket with the following command:

```
$ aws s3api put-bucket-policy --bucket your-app-name-assets --policy
file:///path/to/your/Desktop/policy.json
```

Configure Your App

The next step is to configure your app's uploader to use S3, rather than local file storage. When dealing with uploads, I prefer to use the popular [Carrierwave](#) gem, but the setup will be similar with uploaders such as [Paperclip](#) and [Dragonfly](#).

Add Carrierwave and fog to your app's Gemfile. [Fog](#) is the gem used by Carrierwave to communicate with remote services, such as S3:

```
# Gemfile
gem 'carrierwave', '~> 1.1'
gem 'fog-aws', '~> 1.3.0'
gem 'mini_magick', '~> 4.7.0'
```

```
$ bundle install
```

Next, create an initializer for Carrierwave in `config/initializers/carrierwave.rb`:

```
CarrierWave.configure do |config|
  config.fog_provider = 'fog/aws'
  config.fog_credentials = {
    provider:          'AWS',
    aws_access_key_id: ENV.fetch('S3_ACCESS_KEY_ID', ''),
    aws_secret_access_key: ENV.fetch('S3_SECRET_ACCESS_KEY', ''),
    region:            'eu-west-1' # Remember to set this to the
same as your S3 bucket
  }
  config.fog_directory = ENV.fetch('S3_BUCKET_NAME', '')
  config.fog_public     = true
  config.fog_use_ssl_for_aws = true
  config.fog_attributes = { 'Cache-Control' => "max-age=#
{365.day.to_i}" }

  config.storage = :fog
end
```

Note Using `ENV.fetch()` allows us to supply a default value (in this case, an empty string `''`) if the environment variable is not configured. This will allow tasks such as asset precompilation (`bin/rails assets:precompile`) to run without needing to specify the S3 credentials.

Now you can create a Carrierwave *uploader* and mount it into your model. In the example below, we'll create an uploader for a user avatar, and mount it into a **User** model:

```
$ bin/rails g uploader avatar
```

Remove the placeholder content that Carrierwave generates, leaving just the following:

```
# app/uploaders/avatar_upload.rb
class AvatarUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick

  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Resize to 200x200
  process resize_to_fill: [200, 200]

  # Create different versions of your uploaded files:
  version :thumb do
    process resize_to_fit: [60, 60]
  end

  def extension_whitelist
    %w(jpg jpeg png)
  end
end
```

This uploader will resize any uploads to 200x200 pixels, and create a thumbnail at 60x60 pixels. Let's create the user model in which to mount the uploader:

```
$ bin/rails g model user email:string name:string avatar:string
$ bin/rails db:migrate
```

Next, update the generated *user.rb* model to the following:


```
# app/models/user.rb
class User < ApplicationRecord
  validates :email, presence: true
  validates :name, presence: true
  validates :avatar, presence: true

  mount_uploader :avatar, AvatarUploader # Mount your uploader to the
  avatar attribute
end
```

Finally, we'll generate a simple CRUD UI for creating a new user record:

```
$ bin/rails g controller users index new
```

```
# app/views/users/index.html.erb
<h1>Users</h1>

<%= link_to 'Add User', new_user_path %>

<table>
  <% @users.each do |u| %>
    <tr>
      <td><%= image_tag u.avatar.url(:thumb) %></td>
      <td><%= u.name %></td>
      <td><%= link_to 'Delete', user_path(u), method: 'delete' %>
    </td>
  </tr>
<% end %>
</table>
```

```
# app/views/users/new.html.erb
<h1>New User</h1>

<%= form_with(model: @user) do |f| %>
  <p>
    <%= f.label :email %>
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :name %>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :avatar %>
    <%= f.file_field :avatar %>
  </p>
  <%= f.submit %>
end
```

```
<% end %>
```

```
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  def index
    @users = User.order('name').all
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)

    if @user.save
      flash['notice'] = 'User added'
      redirect_to users_url
    else
      flash['alert'] = 'Error adding user'
      render 'new'
    end
  end

  def destroy
    User.find(params[:id]).destroy
    flash['notice'] = 'User deleted'

    redirect_to users_url
  end

  private

  def user_params
    params.require(:user).permit(:email, :name, :avatar)
  end
end
```

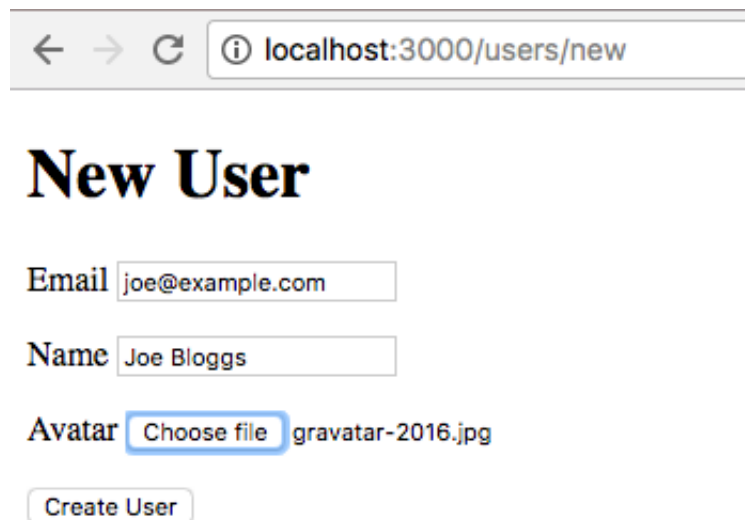
```
# config/routes.rb
Rails.application.routes.draw do
  resources :users, only: [:index, :new, :create]
  # ...
end
```

With everything wired up, it's time to start your app. Remember that you'll need to supply the `S3_ACCESS_KEY_ID`, `S3_SECRET_ACCESS_KEY` and `S3_BUCKET_NAME` environment variables.

For now, you can supply these values directly on the command line, but you'll probably want to use an `.env` file or similar, especially when running your app in production (see [Configuring Application Secrets](#) for details on how to do this)

```
$ S3_ACCESS_KEY_ID=abcdef1234567/abcdefghIJKLmn0pqRStUV  
S3_SECRET_ACCESS_KEY=ABCD1234DEFG56789 S3_BUCKET_NAME=your-app-name-  
assets bin/rails server
```

Open the users admin screen (<http://localhost:3000/users> if you're using the example above) and create a new user account, choosing an image file to upload as the user's avatar.



← → ↻ ⓘ localhost:3000/users/new

New User

Email

Name

Avatar gravatar-2016.jpg

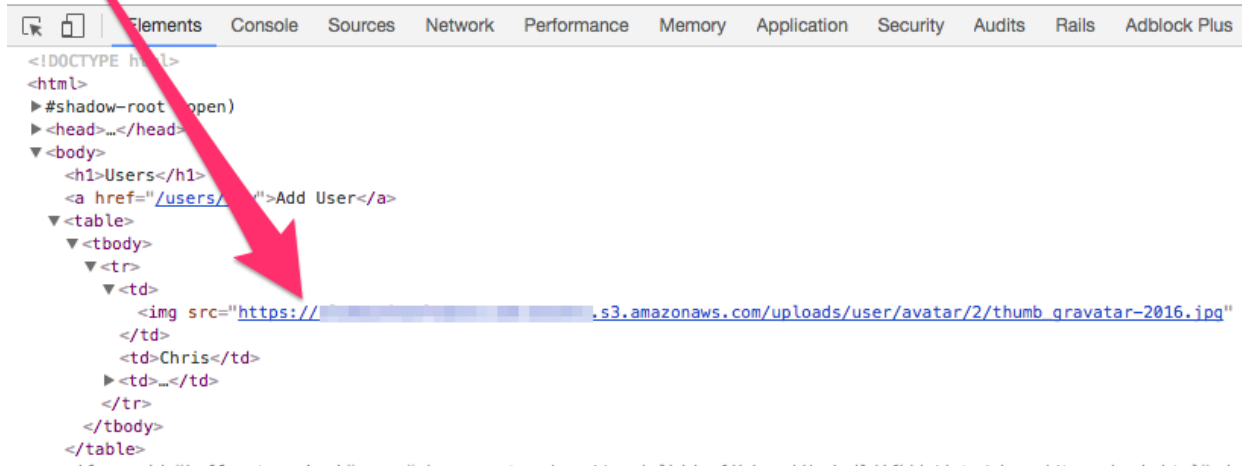
When the save finishes, you'll see your avatar image in the list:

Users

[Add User](#)



Chris [Delete](#)



If you inspect the URL for that image, you'll see that it's served directly from S3 (` | Web | [REDACTED] | [REDACTED].cloudfront.net |

Configuring Carrierwave for CDN

Finally, we just need to tell Carrierwave (and fog) to use the CloudFront CDN URL when displaying uploaded files. This is easy to do with an extra configuration option specifying the Domain CloudFront gave us in the previous step:

```
# config/initializers/carrierwave.rb
CarrierWave.configure do |config|
  # ...
  config.asset_host = 'https://a123456cdefg789.cloudfront.net'
end
```

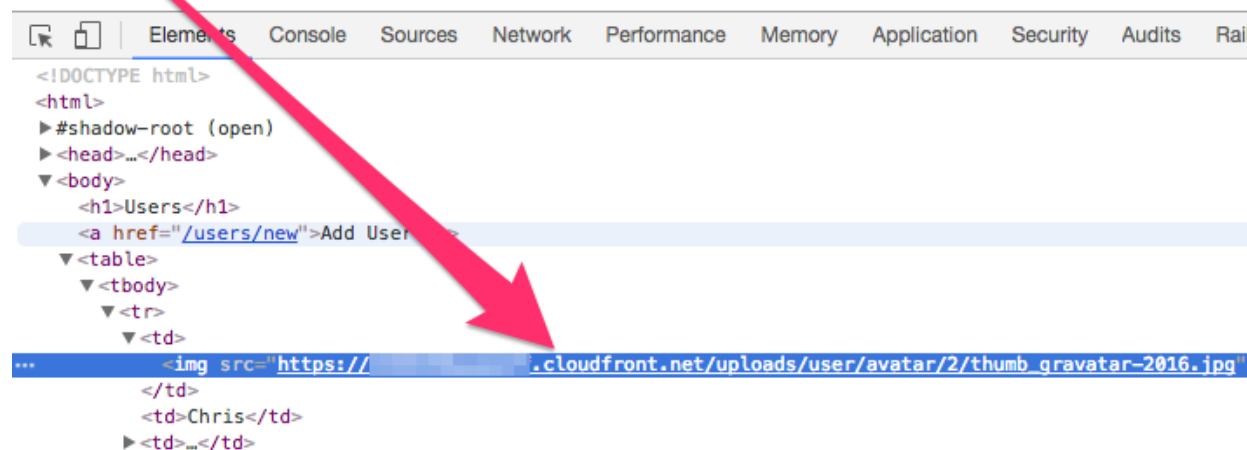
With that done, restart your app (remember to supply the environment variables), and you'll find images are served via CloudFront rather than directly from your S3 bucket:

Users

[Add User](#)



Chris [Delete](#)



Enjoyed this sample?

Purchase the full course today and receive 17 lessons (over 30,000 words) on keeping your Rails apps up and running in top condition.

You'll receive the course in PDF, Kindle (.mobi) and .ePub formats.

Save 10% when you purchase using the promo code `SAMPLEUP10`

Buy Now - Save 10%