**DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.**

# Active Storage Overview

This guide covers how to attach files to your Active Record models.

After reading this guide, you will know:

- How to attach one or many files to a record.
- How to delete an attached file.
- How to link to an attached file.
- How to use variants to transform images.
- How to generate an image representation of a non-image file, such as a PDF or a video.
- How to send file uploads directly from browsers to a storage service, bypassing your application servers.
- How to clean up files stored during testing.
- How to implement support for additional storage services.

---

## What is Active Storage?

Active Storage facilitates uploading files to a cloud storage service like Amazon S3, Google Cloud Storage, or Microsoft Azure Storage and attaching those files to Active Record objects. It comes with a local disk-based service for development and testing and supports mirroring files to subordinate services for backups and migrations.

Using Active Storage, an application can transform image uploads or generate image representations of non-image uploads like PDFs and videos, and extract metadata from arbitrary files.

### Requirements

Various features of Active Storage depend on third-party software which Rails will not install, and must be installed separately:

- libvips v8.6+ or ImageMagick for image analysis and transformations
- ffmpeg v3.4+ for video previews and ffprobe for video/audio analysis
- poppler or muPDF for PDF previews

Image analysis and transformations also require the `image_processing` gem. Uncomment it in your `Gemfile`, or add it if necessary:

```
gem "image_processing", ">= 1.2"
```

TIP: Compared to libvips, ImageMagick is better known and more widely available. However, libvips can be up to 10x faster and consume 1/10 the memory. For JPEG files, this can be further improved by replacing `libjpeg-dev` with `libjpeg-turbo-dev`, which is 2-7x faster.

WARNING: Before you install and use third-party software, make sure you understand the licensing implications of doing so. MuPDF, in particular, is licensed under AGPL and requires a commercial license for some use.

## Setup

Active Storage uses three tables in your application's database named `active_storage_blobs`, `active_storage_variant_records` and `active_storage_attachments`. After creating a new application (or upgrading your application to Rails 5.2), run `bin/rails active_storage:install` to generate a migration that creates these tables. Use `bin/rails db:migrate` to run the migration.

WARNING: `active_storage_attachments` is a polymorphic join table that stores your model's class name. If your model's class name changes, you will need to run a migration on this table to update the underlying `record_type` to your model's new class name.

WARNING: If you are using UUIDs instead of integers as the primary key on your models you will need to change the column type of `active_storage_attachments.record_id` and `active_storage_variant_records.id` in the generated migration accordingly.

Declare Active Storage services in `config/storage.yml`. For each service your application uses, provide a name and the requisite configuration. The example below declares three services named `local`, `test`, and `amazon`:

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

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

amazon:
  service: S3
  access_key_id: ""
  secret_access_key: ""
  bucket: ""
  region: "" # e.g. 'us-east-1'
```

Tell Active Storage which service to use by setting `Rails.application.config.active_storage.service`. Because each environment will likely use a different service, it is recommended to do this on a per-environment basis. To use the disk service from the previous

example in the development environment, you would add the following to `config/environments/development.rb`:

```
# Store files locally.
config.active_storage.service = :local
```

To use the S3 service in production, you add the following to `config/environments/production.rb`:

```
# Store files on Amazon S3.
config.active_storage.service = :amazon
```

To use the test service when testing, you add the following to `config/environments/test.rb`:

```
# Store uploaded files on the local file system in a temporary directory.
config.active_storage.service = :test
```

Continue reading for more information on the built-in service adapters (e.g. `Disk` and `S3`) and the configuration they require.

NOTE: Configuration files that are environment-specific will take precedence: in production, for example, the `config/storage/production.yml` file (if existent) will take precedence over the `config/storage.yml` file.

It is recommended to use `Rails.env` in the bucket names to further reduce the risk of accidentally destroying production data.

```
amazon:
  service: S3
  # ...
  bucket: your_own_bucket-<%= Rails.env %>

google:
  service: GCS
  # ...
  bucket: your_own_bucket-<%= Rails.env %>

azure:
  service: AzureStorage
  # ...
  container: your_container_name-<%= Rails.env %>
```

### Disk Service

Declare a Disk service in `config/storage.yml`:

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

### S3 Service (Amazon S3 and S3-compatible APIs)

To connect to Amazon S3, declare an S3 service in `config/storage.yml`:

```yaml
amazon:
  service: S3
  access_key_id: ""
  secret_access_key: ""
  region: ""
  bucket: ""
```

Optionally provide client and upload options:

```yaml
amazon:
  service: S3
  access_key_id: ""
  secret_access_key: ""
  region: ""
  bucket: ""
  http_open_timeout: 0
  http_read_timeout: 0
  retry_limit: 0
  upload:
    server_side_encryption: "" # 'aws:kms' or 'AES256'
```

TIP: Set sensible client HTTP timeouts and retry limits for your application. In certain failure scenarios, the default AWS client configuration may cause connections to be held for up to several minutes and lead to request queuing.

Add the `aws-sdk-s3` gem to your `Gemfile`:

```
gem "aws-sdk-s3", require: false
```

NOTE: The core features of Active Storage require the following permissions: `s3:ListBucket`, `s3:PutObject`, `s3:GetObject`, and `s3:DeleteObject`. Public access additionally requires `s3:PutObjectAcl`. If you have additional upload options configured such as setting ACLs then additional permissions may be required.

NOTE: If you want to use environment variables, standard SDK configuration files, profiles, IAM instance profiles or task roles, you can omit the `access_key_id`, `secret_access_key`, and `region` keys in the example above. The S3 Service supports all of the authentication options described in the AWS SDK documentation.

To connect to an S3-compatible object storage API such as DigitalOcean Spaces, provide the `endpoint`:

```yaml
digitalocean:
  service: S3
  endpoint: https://nyc3.digitaloceanspaces.com
```

```
  access_key_id: ...
  secret_access_key: ...
  # ...and other options
```

There are many other options available. You can check them in AWS S3 Client documentation.

### Microsoft Azure Storage Service

Declare an Azure Storage service in `config/storage.yml`:

```
azure:
  service: AzureStorage
  storage_account_name: ""
  storage_access_key: ""
  container: ""
```

Add the `azure-storage-blob` gem to your `Gemfile`:

```
gem "azure-storage-blob", require: false
```

### Google Cloud Storage Service

Declare a Google Cloud Storage service in `config/storage.yml`:

```
google:
  service: GCS
  credentials: <%= Rails.root.join("path/to/keyfile.json") %>
  project: ""
  bucket: ""
```

Optionally provide a Hash of credentials instead of a keyfile path:

```
google:
  service: GCS
  credentials:
    type: "service_account"
    project_id: ""
    private_key_id: <%= Rails.application.credentials.dig(:gcs, :private_key_id) %>
    private_key: <%= Rails.application.credentials.dig(:gcs, :private_key).dump %>
    client_email: ""
    client_id: ""
    auth_uri: "https://accounts.google.com/o/oauth2/auth"
    token_uri: "https://accounts.google.com/o/oauth2/token"
    auth_provider_x509_cert_url: "https://www.googleapis.com/oauth2/v1/certs"
    client_x509_cert_url: ""
  project: ""
  bucket: ""
```

Optionally provide a Cache-Control metadata to set on uploaded assets:

```yaml
google:
  service: GCS
  ...
  cache_control: "public, max-age=3600"
```

Optionally use IAM instead of the `credentials` when signing URLs. This is useful if you are authenticating your GKE applications with Workload Identity, see this Google Cloud blog post for more information.

```yaml
google:
  service: GCS
  ...
  iam: true
```

Optionally use a specific GSA when signing URLs. When using IAM, the metadata server will be contacted to get the GSA email, but this metadata server is not always present (e.g. local tests) and you may wish to use a non-default GSA.

```yaml
google:
  service: GCS
  ...
  iam: true
  gsa_email: "foobar@baz.iam.gserviceaccount.com"
```

Add the `google-cloud-storage` gem to your `Gemfile`:

```
gem "google-cloud-storage", "~> 1.11", require: false
```

### Mirror Service

You can keep multiple services in sync by defining a mirror service. A mirror service replicates uploads and deletes across two or more subordinate services.

A mirror service is intended to be used temporarily during a migration between services in production. You can start mirroring to a new service, copy pre-existing files from the old service to the new, then go all-in on the new service.

NOTE: Mirroring is not atomic. It is possible for an upload to succeed on the primary service and fail on any of the subordinate services. Before going all-in on a new service, verify that all files have been copied.

Define each of the services you'd like to mirror as described above. Reference them by name when defining a mirror service:

```yaml
s3_west_coast:
  service: S3
  access_key_id: ""
  secret_access_key: ""
  region: ""
  bucket: ""
```

```yaml
s3_east_coast:
  service: S3
  access_key_id: ""
  secret_access_key: ""
  region: ""
  bucket: ""

production:
  service: Mirror
  primary: s3_east_coast
  mirrors:
    - s3_west_coast
```

Although all secondary services receive uploads, downloads are always handled by the primary service.

Mirror services are compatible with direct uploads. New files are directly uploaded to the primary service. When a directly-uploaded file is attached to a record, a background job is enqueued to copy it to the secondary services.

### Public access

By default, Active Storage assumes private access to services. This means generating signed, single-use URLs for blobs. If you'd rather make blobs publicly accessible, specify `public: true` in your app's `config/storage.yml`:

```yaml
gcs: &gcs
  service: GCS
  project: ""

private_gcs:
  <<: *gcs
  credentials: <%= Rails.root.join("path/to/private_keyfile.json") %>
  bucket: ""

public_gcs:
  <<: *gcs
  credentials: <%= Rails.root.join("path/to/public_keyfile.json") %>
  bucket: ""
  public: true
```

Make sure your buckets are properly configured for public access. See docs on how to enable public read permissions for Amazon S3, Google Cloud Storage, and Microsoft Azure storage services. Amazon S3 additionally requires that you have the `s3:PutObjectAcl` permission.

When converting an existing application to use `public: true`, make sure to

update every individual file in the bucket to be publicly-readable before switching over.

## Attaching Files to Records

**has_one_attached**

The `has_one_attached` macro sets up a one-to-one mapping between records and files. Each record can have one file attached to it.

For example, suppose your application has a `User` model. If you want each user to have an avatar, define the `User` model as follows:

```
class User < ApplicationRecord
  has_one_attached :avatar
end
```

or if you are using Rails 6.0+, you can run a model generator command like this:

```
bin/rails generate model User avatar:attachment
```

You can create a user with an avatar:

```
<%= form.file_field :avatar %>
```

```
class SignupController < ApplicationController
  def create
    user = User.create!(user_params)
    session[:user_id] = user.id
    redirect_to root_path
  end

  private
    def user_params
      params.require(:user).permit(:email_address, :password, :avatar)
    end
end
```

Call `avatar.attach` to attach an avatar to an existing user:

```
user.avatar.attach(params[:avatar])
```

Call `avatar.attached?` to determine whether a particular user has an avatar:

```
user.avatar.attached?
```

In some cases you might want to override a default service for a specific attachment. You can configure specific services per attachment using the `service` option:

```
class User < ApplicationRecord
  has_one_attached :avatar, service: :s3
end
```

You can configure specific variants per attachment by calling the `variant` method on yielded attachable object:

```
class User < ApplicationRecord
  has_one_attached :avatar do |attachable|
    attachable.variant :thumb, resize_to_limit: [100, 100]
  end
end
```

Call `avatar.variant(:thumb)` to get a thumb variant of an avatar:

```
<%= image_tag user.avatar.variant(:thumb) %>
```

#### has_many_attached

The `has_many_attached` macro sets up a one-to-many relationship between records and files. Each record can have many files attached to it.

For example, suppose your application has a `Message` model. If you want each message to have many images, define the `Message` model as follows:

```
class Message < ApplicationRecord
  has_many_attached :images
end
```

or if you are using Rails 6.0+, you can run a model generator command like this:

```
bin/rails generate model Message images:attachments
```

You can create a message with images:

```
class MessagesController < ApplicationController
  def create
    message = Message.create!(message_params)
    redirect_to message
  end

  private
    def message_params
      params.require(:message).permit(:title, :content, images: [])
    end
end
```

Call `images.attach` to add new images to an existing message:

```
@message.images.attach(params[:images])
```

Call `images.attached?` to determine whether a particular message has any images:

```
@message.images.attached?
```

Overriding the default service is done the same way as `has_one_attached`, by using the `service` option:

```
class Message < ApplicationRecord
  has_many_attached :images, service: :s3
end
```

Configuring specific variants is done the same way as `has_one_attached`, by calling the `variant` method on the yielded attachable object:

```
class Message < ApplicationRecord
  has_many_attached :images do |attachable|
    attachable.variant :thumb, resize_to_limit: [100, 100]
  end
end
```

### Attaching File/IO Objects

Sometimes you need to attach a file that doesn't arrive via an HTTP request. For example, you may want to attach a file you generated on disk or downloaded from a user-submitted URL. You may also want to attach a fixture file in a model test. To do that, provide a Hash containing at least an open IO object and a filename:

```
@message.images.attach(io: File.open('/path/to/file'), filename: 'file.pdf')
```

When possible, provide a content type as well. Active Storage attempts to determine a file's content type from its data. It falls back to the content type you provide if it can't do that.

```
@message.images.attach(io: File.open('/path/to/file'), filename: 'file.pdf', content_type:
```

You can bypass the content type inference from the data by passing in `identify: false` along with the `content_type`.

```
@message.images.attach(
  io: File.open('/path/to/file'),
  filename: 'file.pdf',
  content_type: 'application/pdf',
  identify: false
)
```

If you don't provide a content type and Active Storage can't determine the file's content type automatically, it defaults to application/octet-stream.

## Removing Files

To remove an attachment from a model, call `purge` on the attachment. If your application is set up to use Active Job, removal can be done in the background instead by calling `purge_later`. Purging deletes the blob and the file from the storage service.

```ruby
# Synchronously destroy the avatar and actual resource files.
user.avatar.purge

# Destroy the associated models and actual resource files async, via Active Job.
user.avatar.purge_later
```

## Serving Files

Active Storage supports two ways to serve files: redirecting and proxying.

WARNING: All Active Storage controllers are publicly accessible by default. The generated URLs are hard to guess, but permanent by design. If your files require a higher level of protection consider implementing Authenticated Controllers.

### Redirect mode

To generate a permanent URL for a blob, you can pass the blob to the `url_for` view helper. This generates a URL with the blob's `signed_id` that is routed to the blob's `RedirectController`

```ruby
url_for(user.avatar)
# => /rails/active_storage/blobs/:signed_id/my-avatar.png
```

The `RedirectController` redirects to the actual service endpoint. This indirection decouples the service URL from the actual one, and allows, for example, mirroring attachments in different services for high-availability. The redirection has an HTTP expiration of 5 minutes.

To create a download link, use the `rails_blob_{path|url}` helper. Using this helper allows you to set the disposition.

```ruby
rails_blob_path(user.avatar, disposition: "attachment")
```

WARNING: To prevent XSS attacks, Active Storage forces the Content-Disposition header to "attachment" for some kind of files. To change this behaviour see the available configuration options in Configuring Rails Applications.

If you need to create a link from outside of controller/view context (Background jobs, Cronjobs, etc.), you can access the `rails_blob_path` like this:

```ruby
Rails.application.routes.url_helpers.rails_blob_path(user.avatar, only_path: true)
```

### Proxy mode

Optionally, files can be proxied instead. This means that your application servers will download file data from the storage service in response to requests. This can be useful for serving files from a CDN.

You can configure Active Storage to use proxying by default:

```ruby
# config/initializers/active_storage.rb
Rails.application.config.active_storage.resolve_model_to_route = :rails_storage_proxy
```

Or if you want to explicitly proxy specific attachments there are URL helpers you can use in the form of `rails_storage_proxy_path` and `rails_storage_proxy_url`.

```
<%= image_tag rails_storage_proxy_path(@user.avatar) %>
```

**Putting a CDN in front of Active Storage** Additionally, in order to use a CDN for Active Storage attachments, you will need to generate URLs with proxy mode so that they are served by your app and the CDN will cache the attachment without any extra configuration. This works out of the box because the default Active Storage proxy controller sets an HTTP header indicating to the CDN to cache the response.

You should also make sure that the generated URLs use the CDN host instead of your app host. There are multiple ways to achieve this, but in general it involves tweaking your `config/routes.rb` file so that you can generate the proper URLs for the attachments and their variations. As an example, you could add this:

```ruby
# config/routes.rb
direct :cdn_image do |model, options|
  expires_in = options.delete(:expires_in) { ActiveStorage.urls_expire_in }

  if model.respond_to?(:signed_id)
    route_for(
      :rails_service_blob_proxy,
      model.signed_id(expires_in: expires_in),
      model.filename,
      options.merge(host: ENV['CDN_HOST'])
    )
  else
    signed_blob_id = model.blob.signed_id(expires_in: expires_in)
    variation_key  = model.variation.key
    filename       = model.blob.filename

    route_for(
      :rails_blob_representation_proxy,
      signed_blob_id,
      variation_key,
      filename,
      options.merge(host: ENV['CDN_HOST'])
    )
  end
end
```

and then generate routes like this:

```
<%= cdn_image_url(user.avatar.variant(resize_to_limit: [128, 128])) %>
```

#### Authenticated Controllers

All Active Storage controllers are publicly accessible by default. The generated URLs use a plain `signed_id`, making them hard to guess but permanent. Anyone that knows the blob URL will be able to access it, even if a `before_action` in your `ApplicationController` would otherwise require a login. If your files require a higher level of protection, you can implement your own authenticated controllers, based on the `ActiveStorage::Blobs::RedirectController`, `ActiveStorage::Blobs::ProxyController`, `ActiveStorage::Representations::RedirectController` and `ActiveStorage::Representations::ProxyController`

To only allow an account to access their own logo you could do the following:

```
# config/routes.rb
resource :account do
  resource :logo
end
```

```
# app/controllers/logos_controller.rb
class LogosController < ApplicationController
  # Through ApplicationController:
  # include Authenticate, SetCurrentAccount

  def show
    redirect_to Current.account.logo.url
  end
end
```

```
<%= image_tag account_logo_path %>
```

And then you might want to disable the Active Storage default routes with:

```
config.active_storage.draw_routes = false
```

to prevent files being accessed with the publicly accessible URLs.

### Downloading Files

Sometimes you need to process a blob after it's uploaded—for example, to convert it to a different format. Use the attachment's `download` method to read a blob's binary data into memory:

```
binary = user.avatar.download
```

You might want to download a blob to a file on disk so an external program (e.g. a virus scanner or media transcoder) can operate on it. Use the attachment's `open` method to download a blob to a tempfile on disk:

```ruby
message.video.open do |file|
  system '/path/to/virus/scanner', file.path
  # ...
end
```

It's important to know that the file is not yet available in the `after_create` callback but in the `after_create_commit` only.

## Analyzing Files

Active Storage analyzes files once they've been uploaded by queuing a job in Active Job. Analyzed files will store additional information in the metadata hash, including `analyzed: true`. You can check whether a blob has been analyzed by calling `analyzed?` on it.

Image analysis provides `width` and `height` attributes. Video analysis provides these, as well as `duration`, `angle`, `display_aspect_ratio`, and `video` and `audio` booleans to indicate the presence of those channels. Audio analysis provides `duration` and `bit_rate` attributes.

## Displaying Images, Videos, and PDFs

Active Storage supports representing a variety of files. You can call `representation` on an attachment to display an image variant, or a preview of a video or PDF. Before calling `representation`, check if the attachment can be represented by calling `representable?`. Some file formats can't be previewed by Active Storage out of the box (e.g. Word documents); if `representable?` returns false you may want to link to the file instead.

```
<ul>
  <% @message.files.each do |file| %>
    <li>
      <% if file.representable? %>
        <%= image_tag file.representation(resize_to_limit: [100, 100]) %>
      <% else %>
        <%= link_to rails_blob_path(file, disposition: "attachment") do %>
          <%= image_tag "placeholder.png", alt: "Download file" %>
        <% end %>
      <% end %>
    </li>
  <% end %>
</ul>
```

Internally, `representation` calls `variant` for images, and `preview` for previewable files. You can also call these methods directly.

### Lazy vs Immediate Loading

By default, Active Storage will process representations lazily. This code:

```
image_tag file.representation(resize_to_limit: [100, 100])
```

Will generate an `<img>` tag with the `src` pointing to the `ActiveStorage::Representations::RedirectControl`
The browser will make a request to that controller, which will return a `302` redirect to the file on the remote service (or in proxy mode, return the file contents). Loading the file lazily allows features like single use URLs to work without slowing down your initial page loads.

This works fine for most cases.

If you want to generate URLs for images immediately, you can call `.processed.url`:

```
image_tag file.representation(resize_to_limit: [100, 100]).processed.url
```

The Active Storage variant tracker improves performance of this, by storing a record in the database if the requested representation has been processed before. Thus, the above code will only make an API call to the remote service (e.g. S3) once, and once a variant is stored, will use that. The variant tracker runs automatically, but can be disabled through `config.active_storage.track_variants`.

If you're rendering lots of images on a page, the above example could result in N+1 queries loading all the variant records. To avoid these N+1 queries, use the named scopes on `ActiveStorage::Attachment`.

```
message.images.with_all_variant_records.each do |file|
  image_tag file.representation(resize_to_limit: [100, 100]).processed.url
end
```

### Transforming Images

Transforming images allows you to display the image at your choice of dimensions. To create a variation of an image, call `variant` on the attachment. You can pass any transformation supported by the variant processor to the method. When the browser hits the variant URL, Active Storage will lazily transform the original blob into the specified format and redirect to its new service location.

```
<%= image_tag user.avatar.variant(resize_to_limit: [100, 100]) %>
```

If a variant is requested, Active Storage will automatically apply transformations depending on the image's format:

1. Content types that are variable (as dictated by `config.active_storage.variable_content_types`) and not considered web images (as dictated by `config.active_storage.web_image_content_types`), will be converted to PNG.
2. If `quality` is not specified, the variant processor's default quality for the format will be used.

Active Storage can use either Vips or MiniMagick as the variant processor. The default depends on your `config.load_defaults` target version, and the processor can be changed by setting `config.active_storage.variant_processor`.

The two processors are not fully compatible, so when migrating an existing application between MiniMagick and Vips, some changes have to be made if using options that are format specific:

```
<!-- MiniMagick -->
<%= image_tag user.avatar.variant(resize_to_limit: [100, 100], format: :jpeg, sampling_fact

<!-- Vips -->
<%= image_tag user.avatar.variant(resize_to_limit: [100, 100], format: :jpeg, saver: { subs
```

### Previewing Files

Some non-image files can be previewed: that is, they can be presented as images. For example, a video file can be previewed by extracting its first frame. Out of the box, Active Storage supports previewing videos and PDF documents. To create a link to a lazily-generated preview, use the attachment's `preview` method:

```
<%= image_tag message.video.preview(resize_to_limit: [100, 100]) %>
```

To add support for another format, add your own previewer. See the `ActiveStorage::Preview` documentation for more information.

## Direct Uploads

Active Storage, with its included JavaScript library, supports uploading directly from the client to the cloud.

### Usage

1. Include `activestorage.js` in your application's JavaScript bundle.

   Using the asset pipeline:

   ```
   //= require activestorage
   ```

   Using the npm package:

   ```
   import * as ActiveStorage from "@rails/activestorage"
   ActiveStorage.start()
   ```

2. Add `direct_upload: true` to your file field:

   ```
   <%= form.file_field :attachments, multiple: true, direct_upload: true %>
   ```

   Or, if you aren't using a `FormBuilder`, add the data attribute directly:

   ```
   <input type=file data-direct-upload-url="<%= rails_direct_uploads_url %>" />
   ```

3. Configure CORS on third-party storage services to allow direct upload requests.
4. That's it! Uploads begin upon form submission.

### Cross-Origin Resource Sharing (CORS) configuration

To make direct uploads to a third-party service work, you'll need to configure the service to allow cross-origin requests from your app. Consult the CORS documentation for your service:

- S3
- Google Cloud Storage
- Azure Storage

Take care to allow:

- All origins from which your app is accessed
- The `PUT` request method
- The following headers:
  - `Origin`
  - `Content-Type`
  - `Content-MD5`
  - `Content-Disposition` (except for Azure Storage)
  - `x-ms-blob-content-disposition` (for Azure Storage only)
  - `x-ms-blob-type` (for Azure Storage only)
  - `Cache-Control` (for GCS, only if `cache_control` is set)

No CORS configuration is required for the Disk service since it shares your app's origin.

### Example: S3 CORS configuration

```
[
  {
    "AllowedHeaders": [
      "*"
    ],
    "AllowedMethods": [
      "PUT"
    ],
    "AllowedOrigins": [
      "https://www.example.com"
    ],
    "ExposeHeaders": [
      "Origin",
      "Content-Type",
      "Content-MD5",
      "Content-Disposition"
```

```
    ],
    "MaxAgeSeconds": 3600
  }
]
```

**Example: Google Cloud Storage CORS configuration**

```
[
  {
    "origin": ["https://www.example.com"],
    "method": ["PUT"],
    "responseHeader": ["Origin", "Content-Type", "Content-MD5", "Content-Disposition"],
    "maxAgeSeconds": 3600
  }
]
```

**Example: Azure Storage CORS configuration**

```
<Cors>
  <CorsRule>
    <AllowedOrigins>https://www.example.com</AllowedOrigins>
    <AllowedMethods>PUT</AllowedMethods>
    <AllowedHeaders>Origin, Content-Type, Content-MD5, x-ms-blob-content-disposition, x-ms-b
    <MaxAgeInSeconds>3600</MaxAgeInSeconds>
  </CorsRule>
</Cors>
```

**Direct upload JavaScript events**

| Event name | Event target | Event data (`event.detail`) | Description |
|---|---|---|---|
| `direct-uploads:start` | `<form>` | None | A form containing files for direct upload fields was submitted. |
| `direct-upload:initialize` | `<input>` | `{id, file}` | Dispatched for every file after form submission. |
| `direct-upload:start` | `<input>` | `{id, file}` | A direct upload is starting. |
| `direct-upload:before-blob-request` | `<input>` | `{id, file, xhr}` | Before making a request to your application for direct upload metadata. |

| Event name | Event target | Event data (`event.detail`) | Description |
|---|---|---|---|
| `direct-upload:before-storage-request` | `<input>` | `{id, file, xhr}` | Before making a request to store a file. |
| `direct-upload:progress` | `<input>` | `{id, file, progress}` | As requests to store files progress. |
| `direct-upload:error` | `<input>` | `{id, file, error}` | An error occurred. An `alert` will display unless this event is canceled. |
| `direct-upload:end` | `<input>` | `{id, file}` | A direct upload has ended. |
| `direct-uploads:end` | `<form>` | None | All direct uploads have ended. |

### Example

You can use these events to show the progress of an upload.

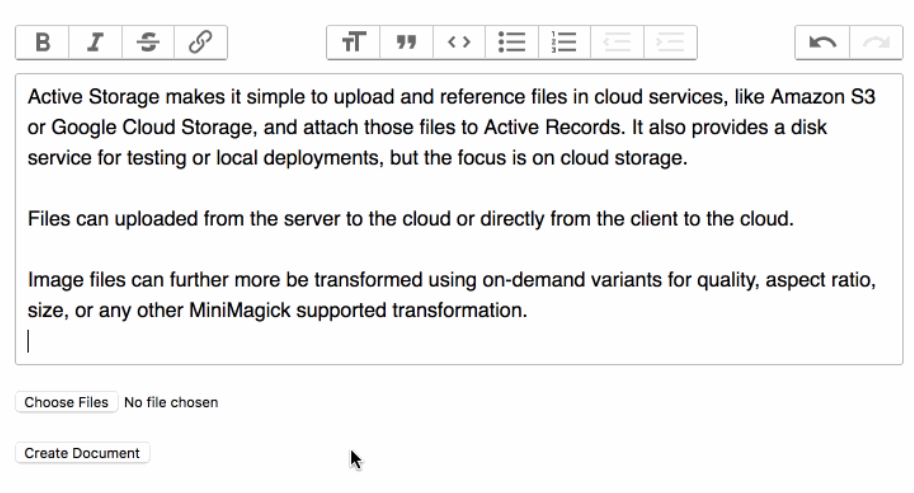


Figure 1: direct-uploads

To show the uploaded files in a form:

```
// direct_uploads.js

addEventListener("direct-upload:initialize", event => {
```

```
  const { target, detail } = event
  const { id, file } = detail
  target.insertAdjacentHTML("beforebegin", `
    <div id="direct-upload-${id}" class="direct-upload direct-upload--pending">
      <div id="direct-upload-progress-${id}" class="direct-upload__progress" style="width: 0
      <span class="direct-upload__filename"></span>
    </div>
  `)
  target.previousElementSibling.querySelector(`.direct-upload__filename`).textContent = file
})

addEventListener("direct-upload:start", event => {
  const { id } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.remove("direct-upload--pending")
})

addEventListener("direct-upload:progress", event => {
  const { id, progress } = event.detail
  const progressElement = document.getElementById(`direct-upload-progress-${id}`)
  progressElement.style.width = `${progress}%`
})

addEventListener("direct-upload:error", event => {
  event.preventDefault()
  const { id, error } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.add("direct-upload--error")
  element.setAttribute("title", error)
})

addEventListener("direct-upload:end", event => {
  const { id } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.add("direct-upload--complete")
})
```

Add styles:

```
/* direct_uploads.css */

.direct-upload {
  display: inline-block;
  position: relative;
  padding: 2px 4px;
  margin: 0 3px 3px 0;
  border: 1px solid rgba(0, 0, 0, 0.3);
```

```
  border-radius: 3px;
  font-size: 11px;
  line-height: 13px;
}

.direct-upload--pending {
  opacity: 0.6;
}

.direct-upload__progress {
  position: absolute;
  top: 0;
  left: 0;
  bottom: 0;
  opacity: 0.2;
  background: #0076ff;
  transition: width 120ms ease-out, opacity 60ms 60ms ease-in;
  transform: translate3d(0, 0, 0);
}

.direct-upload--complete .direct-upload__progress {
  opacity: 0.4;
}

.direct-upload--error {
  border-color: red;
}

input[type=file][data-direct-upload-url][disabled] {
  display: none;
}
```

### Integrating with Libraries or Frameworks

If you want to use the Direct Upload feature from a JavaScript framework, or you want to integrate custom drag and drop solutions, you can use the `DirectUpload` class for this purpose. Upon receiving a file from your library of choice, instantiate a DirectUpload and call its create method. Create takes a callback to invoke when the upload completes.

```
import { DirectUpload } from "@rails/activestorage"

const input = document.querySelector('input[type=file]')

// Bind to file drop - use the ondrop on a parent element or use a
//  library like Dropzone
```

```
const onDrop = (event) => {
  event.preventDefault()
  const files = event.dataTransfer.files;
  Array.from(files).forEach(file => uploadFile(file))
}

// Bind to normal file selection
input.addEventListener('change', (event) => {
  Array.from(input.files).forEach(file => uploadFile(file))
  // you might clear the selected files from the input
  input.value = null
})

const uploadFile = (file) => {
  // your form needs the file_field direct_upload: true, which
  //  provides data-direct-upload-url
  const url = input.dataset.directUploadUrl
  const upload = new DirectUpload(file, url)

  upload.create((error, blob) => {
    if (error) {
      // Handle the error
    } else {
      // Add an appropriately-named hidden input to the form with a
      //  value of blob.signed_id so that the blob ids will be
      //  transmitted in the normal upload flow
      const hiddenField = document.createElement('input')
      hiddenField.setAttribute("type", "hidden");
      hiddenField.setAttribute("value", blob.signed_id);
      hiddenField.name = input.name
      document.querySelector('form').appendChild(hiddenField)
    }
  })
}
```

If you need to track the progress of the file upload, you can pass a third parameter to the `DirectUpload` constructor. During the upload, DirectUpload will call the object's `directUploadWillStoreFileWithXHR` method. You can then bind your own progress handler on the XHR.

```
import { DirectUpload } from "@rails/activestorage"

class Uploader {
  constructor(file, url) {
    this.upload = new DirectUpload(this.file, this.url, this)
  }
```

```
  upload(file) {
    this.upload.create((error, blob) => {
      if (error) {
        // Handle the error
      } else {
        // Add an appropriately-named hidden input to the form
        // with a value of blob.signed_id
      }
    })
  }

  directUploadWillStoreFileWithXHR(request) {
    request.upload.addEventListener("progress",
      event => this.directUploadDidProgress(event))
  }

  directUploadDidProgress(event) {
    // Use event.loaded and event.total to update the progress bar
  }
}
```

NOTE: Using Direct Uploads can sometimes result in a file that uploads, but never attaches to a record. Consider purging unattached uploads.

## Testing

Use `fixture_file_upload` to test uploading a file in an integration or controller test. Rails handles files like any other parameter.

```
class SignupController < ActionDispatch::IntegrationTest
  test "can sign up" do
    post signup_path, params: {
      name: "David",
      avatar: fixture_file_upload("david.png", "image/png")
    }

    user = User.order(:created_at).last
    assert user.avatar.attached?
  end
end
```

### Discarding files created during tests

**System tests** System tests clean up test data by rolling back a transaction. Because `destroy` is never called on an object, the attached files are never cleaned up. If you want to clear the files, you can do it in an `after_teardown` callback. Doing it here ensures that all connections created during the test are complete

and you won't receive an error from Active Storage saying it can't find a file.

```
class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  # ...
  def after_teardown
    super
    FileUtils.rm_rf(ActiveStorage::Blob.service.root)
  end
  # ...
end
```

If you're using parallel tests and the `DiskService`, you should configure each process to use its own folder for Active Storage. This way, the `teardown` callback will only delete files from the relevant process' tests.

```
class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  # ...
  parallelize_setup do |i|
    ActiveStorage::Blob.service.root = "#{ActiveStorage::Blob.service.root}-#{i}"
  end
  # ...
end
```

If your system tests verify the deletion of a model with attachments and you're using Active Job, set your test environment to use the inline queue adapter so the purge job is executed immediately rather at an unknown time in the future.

```
# Use inline job processing to make things happen immediately
config.active_job.queue_adapter = :inline
```

**Integration tests** Similarly to System Tests, files uploaded during Integration Tests will not be automatically cleaned up. If you want to clear the files, you can do it in an `teardown` callback.

```
class ActionDispatch::IntegrationTest
  def after_teardown
    super
    FileUtils.rm_rf(ActiveStorage::Blob.service.root)
  end
end
```

If you're using parallel tests and the Disk service, you should configure each process to use its own folder for Active Storage. This way, the `teardown` callback will only delete files from the relevant process' tests.

```
class ActionDispatch::IntegrationTest
  parallelize_setup do |i|
    ActiveStorage::Blob.service.root = "#{ActiveStorage::Blob.service.root}-#{i}"
```

```
  end
end
```

### Adding attachments to fixtures

You can add attachments to your existing fixtures. First, you'll want to create a separate storage service:

```
# config/storage.yml

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

This tells Active Storage where to "upload" fixture files to, so it should be a temporary directory. By making it a different directory to your regular `test` service, you can separate fixture files from files uploaded during a test.

Next, create fixture files for the Active Storage classes:

```
# active_storage/attachments.yml
david_avatar:
  name: avatar
  record: david (User)
  blob: david_avatar_blob
```

```
# active_storage/blobs.yml
david_avatar_blob: <%= ActiveStorage::FixtureSet.blob filename: "david.png", service_name: '
```

Then put a file in your fixtures directory (the default path is `test/fixtures/files`) with the corresponding filename. See the `ActiveStorage::FixtureSet` docs for more information.

Once everything is set up, you'll be able to access attachments in your tests:

```
class UserTest < ActiveSupport::TestCase
  def test_avatar
    avatar = users(:david).avatar

    assert avatar.attached?
    assert_not_nil avatar.download
    assert_equal 1000, avatar.byte_size
  end
end
```

**Cleaning up fixtures** While files uploaded in tests are cleaned up at the end of each test, you only need to clean up fixture files once: when all your tests complete.

If you're using parallel tests, call `parallelize_teardown`:

```
class ActiveSupport::TestCase
  # ...
  parallelize_teardown do |i|
    FileUtils.rm_rf(ActiveStorage::Blob.services.fetch(:test_fixtures).root)
  end
  # ...
end
```

If you're not running parallel tests, use `Minitest.after_run` or the equivalent for your test framework (e.g. `after(:suite)` for RSpec):

```
# test_helper.rb

Minitest.after_run do
  FileUtils.rm_rf(ActiveStorage::Blob.services.fetch(:test_fixtures).root)
end
```

### Implementing Support for Other Cloud Services

If you need to support a cloud service other than these, you will need to implement the Service. Each service extends `ActiveStorage::Service` by implementing the methods necessary to upload and download files to the cloud.

### Purging Unattached Uploads

There are cases where a file is uploaded but never attached to a record. This can happen when using Direct Uploads. You can query for unattached records using the unattached scope. Below is an example using a custom rake task.

```
namespace :active_storage do
  desc "Purges unattached Active Storage blobs. Run regularly."
  task purge_unattached: :environment do
    ActiveStorage::Blob.unattached.where("active_storage_blobs.created_at <= ?", 2.days.ago)
  end
end
```

WARNING: The query generated by `ActiveStorage::Blob.unattached` can be slow and potentially disruptive on applications with larger databases.