

RAILS ON AWS



PAWEŁ DĄBROWSKI



iRonin.IT
Software House

MASTERMIND.DEV

Preface

Cloud computing has been one of the hottest topics in the IT industry in the last few years. Given the fact that the importance of artificial intelligence is growing like never before and cloud computing gives us the ability to use services that directly support AI solutions, it's evident that sooner or later, as a software engineer, you would have to deal to some extent with AWS, Azure, GCP or other cloud providers.

Soon, the most successful software engineers will be the ones who act as architects instead of focusing on one or two core technologies, building complex systems, and fulfilling different roles with the help of AI.

The idea of this book came to me as I stood on the edge of the three different roles: software engineer, architect, and DevOps. What is funny is that, many years ago, I didn't like AWS at all. I was a fan of the traditional approach where we store files on the same server as the application instance, build features, not outsource them, and avoid containerization. At some point, I realized I didn't know how to use it properly, and my ego was trying to find excuses and justifications.

As I changed my mindset, everything else has changed. I started to act like a software engineer, not a developer who stuck only to the leading technology. After becoming a CTO of [iRonin - Software House](#), an organization that extensively uses AWS, I decided to master this solution. Currently, I hold four AWS certifications.

I'm not trying to convince you to become a DevOps or AWS expert. I aim to show you how you can utilize AWS as a Rails developer to build bulletproof, secure, reliable, and scalable applications faster. As I worked with hundreds of clients and developers on various projects, I constantly saw that knowledge of AWS is a valuable asset for every Rails developer and helps them find a successful path in the demanding market. This

book covers a minimum amount of theory to understand how things are working and a maximum number of code samples to use immediately in your projects. The information and code presented in this book are frequently updated to keep the content as valuable as possible.

The expertise presented in this book results from hundreds of projects delivered by [iRonin - Software House](#) for companies of different sizes. The code samples are production-tested and designed so you can use them immediately in your projects. I hope you will enjoy reading this book as much as I enjoyed writing it for you. Thank you for your support.

Bug reporting: if you will find a bug in the book, either in the content or the code, please let me know by writing to contact@paweldabrowski.com or contacting me via [Twitter](#). Thank you!

Introduction

You will get the most out of this book by reading it and then using the presented code and solutions in your code. To do this, you need have:

- **Basic Rails application** - Ruby 3.2.2 and Rails 7.1.2
- **AWS account** - you can set up it for free and then use the free tier of services not to produce any costs

I assume you have experience writing Ruby code, at least on the basic level, where you can build elementary Rails applications with a few controllers, models, and some views.

Feel free to jump to "Test Rails application creation" if you already have your AWS account and are familiar with the following terms regarding AWS: users, permissions, services, regions, and access keys.

AWS account creation

Visit <https://portal.aws.amazon.com/billing/signup#/start/email> address and provide the primary e-mail and name for your account:



Explore Free Tier products with a new AWS account.

To learn more, visit aws.amazon.com/free.



Sign up for AWS

Root user email address

Used for account recovery and some administrative functions

AWS account name

Choose a name for your account. You can change this name in your account settings after you sign up.

Verify email address

OR

Sign in to an existing AWS account

You will receive an email with the verification code to the address you provided. Once you provide it, it's time to set up a password for your account. After password setup, you will be asked about your phone number and address. It's required for legal purposes and to avoid scam accounts.

Once you finish this part, you must provide credit card details. Don't worry; as I mentioned before, you will have access to the free tier of services, and it's even mentioned on the form you are currently on. The AWS account itself is free.

Tip: whenever it's possible, I use a virtual credit card with a prepaid balance. It works like a standard credit card; it has a number, CVC code, and expiration date.

Once you add your credit card, you will be asked to confirm your identity by voice or text. I always select text message, type the code I received, select that I don't want any paid support for my account, and it's done.

Congratulations, you just have set up your own AWS account! Sign in and let's create the user we will use for the rest of this book.

User creation

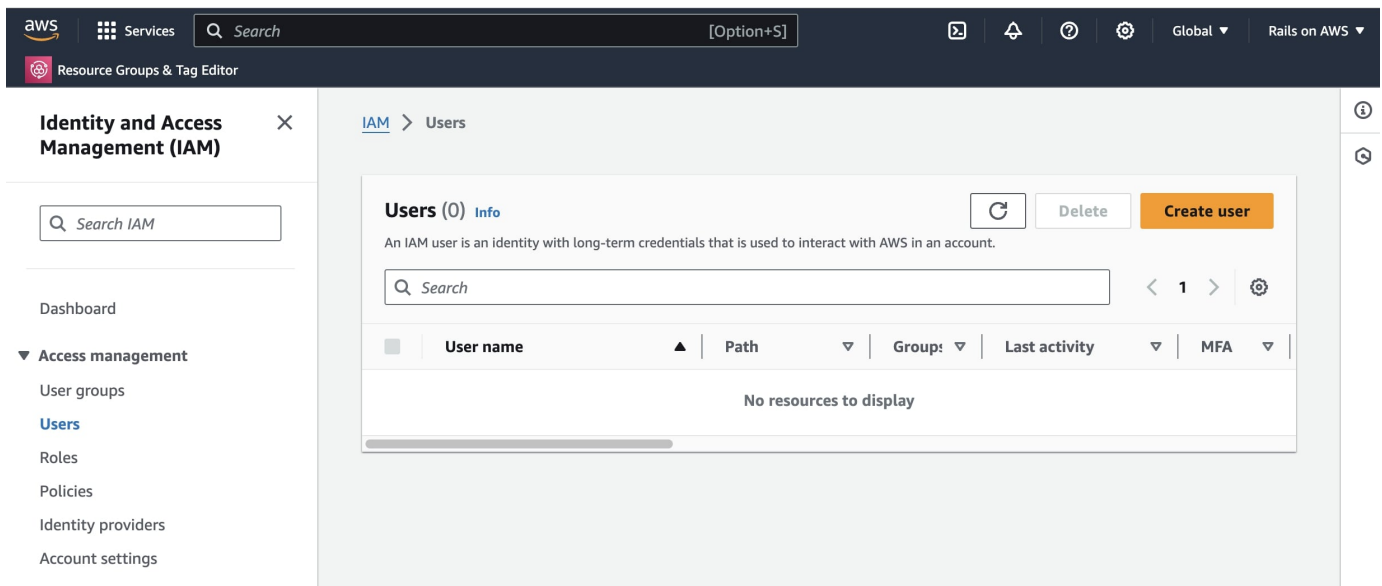
Each user that is using a given AWS account should have separate credentials. You can access all services on the platform as a root account, but we will create a separate account to show you how to add permissions securely.

According to best practices, the root account should not be used for any development or service usage; it should be used only for the primary account setup and never shared with anyone.

Search for **IAM** service, and then on the left sidebar, select the **users** link to access the list of users for your account.

Identity and access management

You should see a blank list that looks similar to this one:



On the top right side of the screen, you will see the account name and "Global". "Global" indicates that you didn't select any region. **Region** is a physical location where

AWS servers are placed. The IAM service is global and free; it manages permissions for different users.

Adding new user

Click on the **Create** user button and provide the user name on the form. I would name my user "test-rails-app" as it only uses AWS API via the test Rails application. This newly created user won't have access to the AWS console via the web interface as you have now.

On the "Set permissions" step, just click "Next", and on the next screen, click "Create user". That's it.

Generating API access keys

We want our "test-rails-app" user to access AWS API, so we must generate proper credentials. Click on the user on the list, and on the detailed view, click on the "Security credentials" tab.

Scroll down to the "Access key" section and click on the "Create access key" button. On the next screen, select "Command Line Interface (CLI)" as a use case and select the confirmation checkbox at the bottom. Click "Next", skip the description input, and click "Create access key".

You can now copy your **Access key** and **Secret access key** or download a CSV file with credentials; remember that you won't see the secret access key again if you don't copy it now. Save the credentials in a safe place; we will need them later. Don't share them with anyone. You can permanently delete them in an emergency, and they will stop working.

Summary and next steps

Let's quickly summarize what we have done so far:

- We created a brand new AWS account for testing
- We created a user for our Rails application
- We created API credentials for our user used by the Rails application

Go ahead and enable 2FA authentication for your root account. You should always do this after setting up a new AWS account. When you set a new user with access to the web interface, enforce 2FA authentication by default to keep the highest security standards.

The last piece that needs to be included is the test Rails application. I will quickly create one with the support for ENV variables to use our API credentials to access AWS services safely.

Test Rails application creation

If you jumped to this section because you already had your AWS account, please create a test user and assign API credentials. Otherwise, if you followed my instructions, you should have a test user already created.

Make sure that you have Ruby 3.2.2 installed and selected. I'm going to generate a new Rails 7.1.2 application with the PostgreSQL database:

```
rails _7.1.2_ new railsonaws -d=postgresql
```

Let's create the database for the application and add the `dotenv-rails` gem to support environment variables in our application:

```
cd railsonaws/  
./bin/rails db:create
```



```
bundle add dotenv-rails -g=development,test
```

Configuration of environment variables

We will store our credentials to AWS in the `.env` file. First, ensure it won't be included in the GIT repository - look at the `.gitignore` file. It should be ignored by default, but it's better to check.

It's also a good idea to create a `.env.example` file that will be included in the repository. Each time someone copies the project, he will know what credentials to include in his copy of the `.env` file.

In the `.env.example` put the following lines:

```
AWS_ACCESS_KEY_ID=  
AWS_SECRET_ACCESS_KEY=  
AWS_REGION=eu-central-1
```

The value of the `AWS_REGION` setting is not secret so we can put it here. I selected `eu-central-1` because it's closest to my physical location, so feel free to choose a different region. Once you pick a region, remember always to select this region when creating or updating services.

Regarding the `.env` file, copy the content of the `.env.example` file but put the real values for access and secret access keys. Open the rails console and verify that `ENV['AWS_ACCESS_KEY_ID']` contains your key.

We are finally ready to start working with the first AWS service inside our Rails application.

Amazon S3

Amazon Simple Storage Service (Amazon S3) is one of the most popular services in the AWS cloud. It's widely used in Rails applications for file storage also, with Active Storage, which has support for S3 by default.

In this chapter, you will learn the minimum required to manage S3 service effectively and implement it in the Rails application. This chapter will include:

- **Explanation** - I will explain in simple words the idea behind the service
- **Use cases** - I will share with you some popular use cases for the service within the Rails application
- **Configuration** - I will walk you through the configuration process of the service to make it ready for your Rails application
- **Pricing** - I will discuss the general pricing for the service as well as different versions of the service (they are priced a little bit differently)
- **Permissions** - I will show you how to properly configure permissions for your user so he can perform only necessary actions
- **Development** - we will write code together to show you how to utilize the S3 service in your Rails application

AWS provides the free tier for S3 service, which consists of 5 GB of storage for the standard storage class. It should be enough for our tests as we will upload only smaller files.

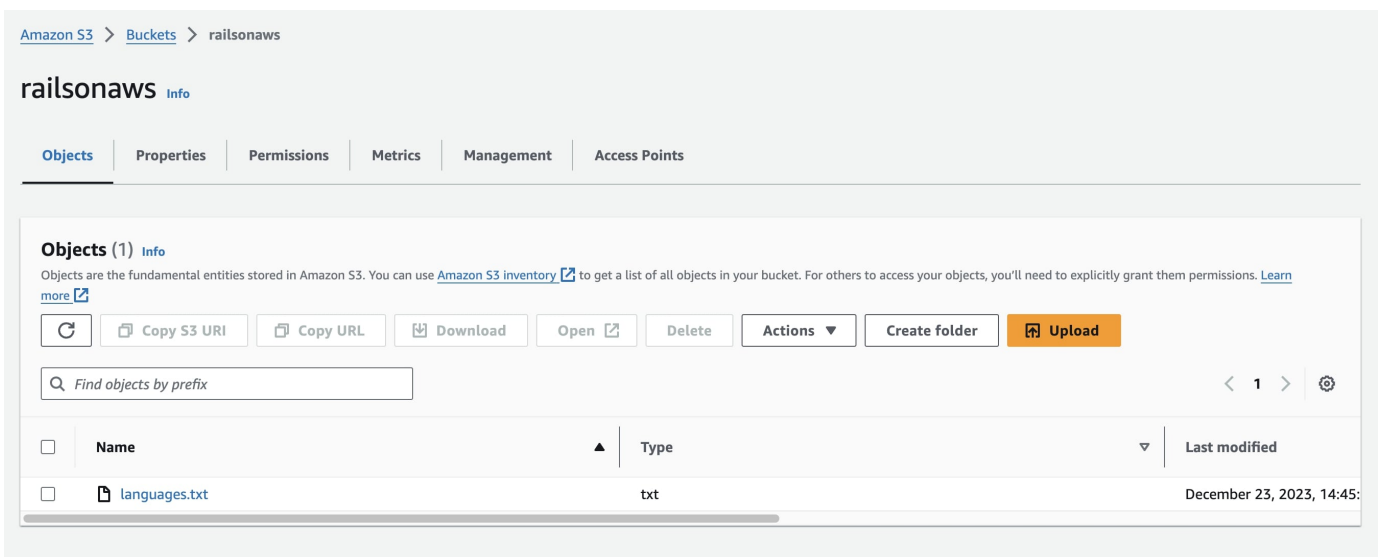
Explanation

Amazon S3 is a service that lets you host your application files and manage them for a very affordable price. Each time you think about storing photos of your users, some reports generated by the application, or any other files uploaded by the system or users themselves - think about the S3.

You can store unlimited files, and a single file can't be bigger than 5 terabytes. The service itself is a flat structure; it consists only of files and names assigned to it. Files

are grouped in buckets. You can have a separate bucket for the staging environment where you keep the avatars of your users and a separate bucket for the production environment where you keep the avatars of your users.

The bucket name **must be unique** across all AWS accounts because it will be visible as part of the URL through which you can access your files. Given that I name my bucket as `railsonaws`, set the region to `eu-central-1`, and upload there a file named `languages.txt`, if I would specify that all files in the bucket are public, I can access my file with the following URL: <https://railsonaws1.s3.eu-central-1.amazonaws.com/languages.txt>



Directories in bucket

You can also create folders inside the bucket but they don't function as ordinary folders you may know from the operating system. As mentioned, S3 is a flat structure, so the folder name becomes part of the file key. If I put the file `languages.txt` into the `text` directory, the key of the file would be `text/languages.txt`, and the URL would look as follows: <https://railsonaws.s3.eu-central-1.amazonaws.com/text/languages.txt>

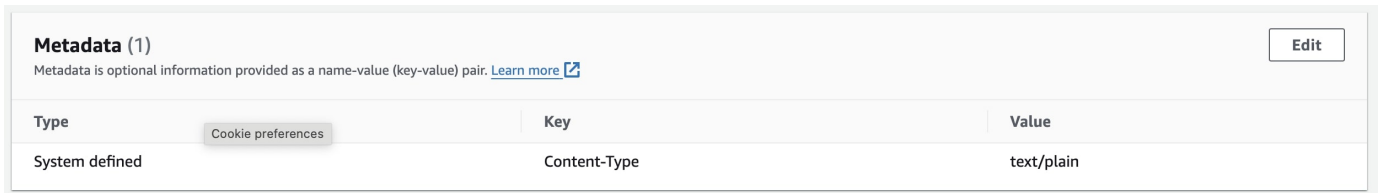
You won't be able to access <https://railsonaws.s3.eu-central-1.amazonaws.com/text> and get a tree of files inside the directory as it would be possible in Linux, Windows, or Mac OS.

Metadata

S3 is a key-value pairs structure where `key` is the name of a file and `value` is the file

content, but you can also attach some additional file information called metadata.

A typical example of metadata is the content type of the file:



Type	Key	Value
System defined	Content-Type	text/plain

You define your entries in the metadata either using the form in the bucket settings or via API when uploading a file.

Storage classes

Amazon introduced the concept of storage classes as we are not using all files the same way:

- **User avatars** - we need to use them frequently; we display them each time someone visits the given user's page
- **Database backups** - we use them from time to time in terms of some disaster cases or debugging process that involve production data
- **Archival data** - we might never use that type of data again, but we need to keep it because of legal rules in the given country

We are okay if we wait a few hours or minutes to download archival data, but it's not acceptable if we pull the user's avatar. In the same way, we are okay with losing some files that we can quickly regenerate at low cost, but losing backup files is unacceptable. Because of the different needs and nature of the files, we have various storage classes at our disposal.

You can view actual storage classes in the official documentation <https://aws.amazon.com/s3/storage-classes/>, but for this book, we will use standard storage, which is used in most of the Rails applications where files are used frequently or from time to time.

Use cases

Below, I collected the list of typical use cases for the S3 service when it comes to the

connection with a Rails application:

- **Attachments storage** - Active Storage in Rails supports S3 by default; to use it, a minimum configuration is required; I will demonstrate it later.
- **Reports storage** - if you have a background job that generates reports, you can store them in an S3 bucket and make them accessible by generating a special pre-signed URL. A pre-signed URL is a unique URL that grants access to the file for a given period.
- **Files versioning** - S3 allows to turn on versioning and keep different versions of the same file
- **Assets hosting** - instead of storing assets in the `app/assets` directory, you can place them in the bucket.

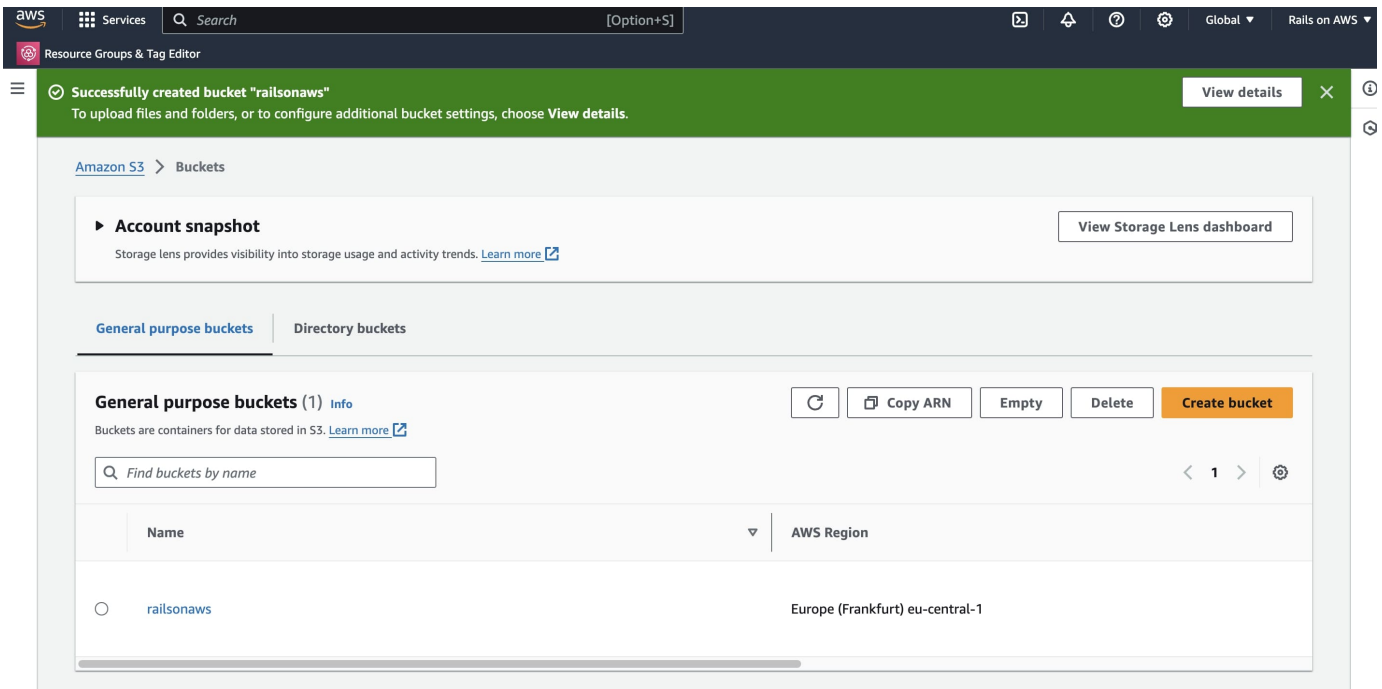
Of course, there are many more use cases. One of the most significant advantages of using S3 over standard server storage is that in the containerized application, you wouldn't be able to access files generated by other application instance without uploading them to the cloud.

Configuration

In this section, we will create the first bucket and place a public file called `languages.txt` containing a list of programming languages. To confirm that our configuration is working, we will access the file via the URL in our browser.

Sign into your AWS account and select the S3 service. Click "Create bucket" on the right-hand side. First, select the region. Each time you create a new bucket, think about the users or servers that will pull files from this bucket - where are they located? Select the closest region, but don't worry if your users are worldwide. There are ways to improve the performance and reduce the latency later.

Give your bucket a unique and meaningful name. You can use a prefix of your company or application to make it unique. Uncheck the "Block all public access" checkbox and confirm your choice; for this configuration, we want all files to be public by default. Click "Create" and you should see the new bucket on the list:



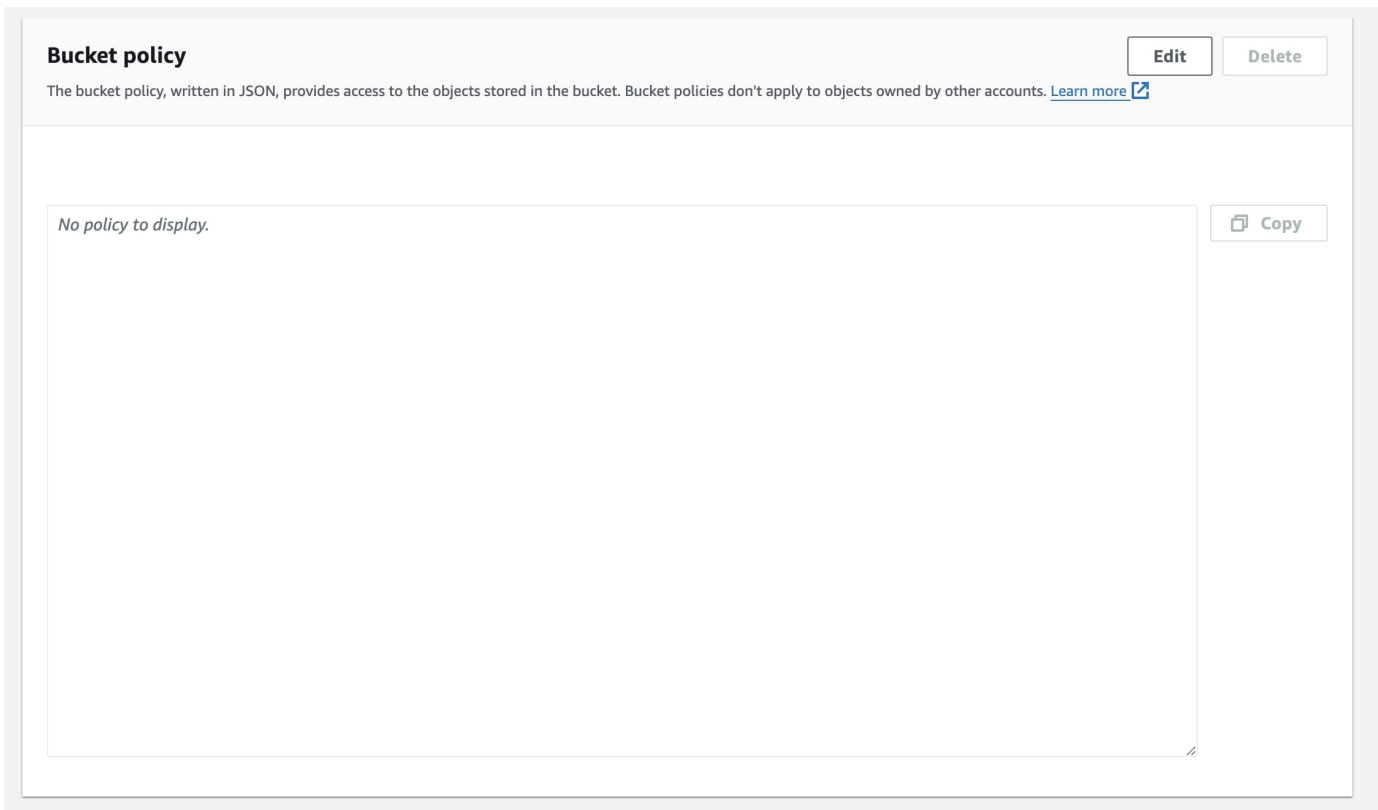
Files upload

Click on the bucket name. You can now upload the file by clicking the "Upload" button or by dragging the file on the list. A confirmation window will appear, and you can click "Upload" without modifying the details.

Click on the file name, and you will be redirected to the file's details page, where you can see information about the file and settings. Under the "Object URL" label, you will find the unique URL that you can use to access the file; click on it. You will receive an error because we need to update bucket permissions.

Bucket permissions

Navigate to the main view of the bucket. You will notice the "Permissions" tab - click on it. Now, scroll down to the bucket policy section and click "Edit":



This is the first time you have to deal with the **AWS Policy**. The policy is a set of rules in JSON format that defines who has access to which elements of the service. In our case, the content of the policy would be the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-name/*"
      ]
    }
  ]
}
```

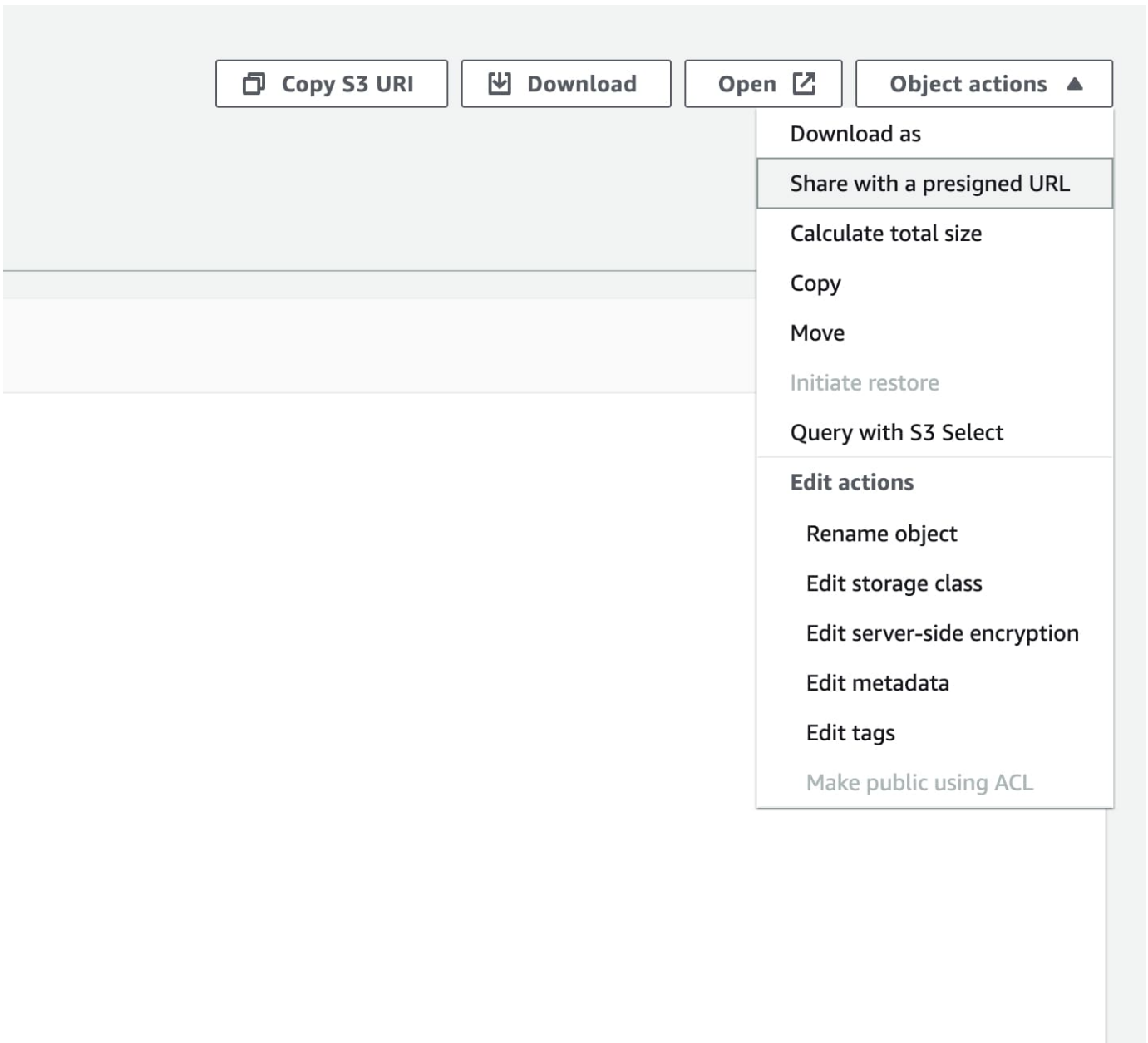
In a shortcut, we allow (Effect attribute) anyone (Principal attribute) to perform action `s3:GetObject` (show the file) in resource `arn:aws:s3:::bucket-name/*` - replace `bucket-name` with the name of your bucket. Paste the policy content with the changed bucket name and save the changes. Now refresh the unique URL to your file - you should be able to see the contents.

I will use different policies throughout the rest of this book and explain them in detail later. The format of policy would be the same; only the resource, action, effect, and principal values would change.

Presigned URLs to files

Delete the bucket policy we set a while ago. If the bucket is not public, we can still access the file by generating a special presigned URL with the expiration time. Everyone with this link can see the file; it's a perfect way to share reports via e-mail.

Navigate to the files list in the bucket and click on our file to see the details of the file. In the right top corner, expand the "Object actions" list and select "Share with a presigned URL":



Specify the number of minutes the link should remain active and submit the form. The unique presigned link is now copied to your clipboard. Paste the link into the browser, and you should be able to read your file.

After the expiration, you won't be able to re-access the file.

Pricing

When using Amazon S3 service, you don't pay only for the storage that you are using. More factors affect the final monthly price for the service usage.

The official pricing page is available at <https://aws.amazon.com/s3/pricing/>, and the price factors are the following:

- **Storage** - you pay for gigabytes stored per month, and the pricing differs depending on the amount of storage you consume. The more you use, the less you pay.
- **Requests** - you pay for every request like PUT, POST, or GET, and the price is for 1,000 requests.
- **Data transfer** - you pay for gigabytes transferred out of the bucket, and the pricing differs depending on the amount of data transferred. The more you transfer, the less you pay.
- **Security access and control** - the base encryption is free; you need to pay for dual-layer server-side encryption or S3 access grants requests (both features are out of the scope of this book)
- **Management and insights** - you need to pay when you use some additional custom features of S3, like tagging millions of files or analytics tools (both features are out of the scope of this book)
- **Replication** - you need to pay for the feature of automatic replication of files to another bucket
- **Transform and query** - you need to pay for requests related to s3 when you use lambda serverless service (you will read more about lambda later)

Unless you have some custom policies in your application or process very complex and unique workloads, all you have to consider regarding s3 pricing is **storage, requests, and data transfer**.

Permissions

You already saw one policy in the bucket configuration section. Permissions are one of the base concepts of the AWS cloud. According to best practices, you should always grant the least amount of permission needed to perform the job. If you need your code to upload files to the S3 bucket, you only allow to upload files to a certain bucket, not to access and manage all files in the bucket.

Broader permissions are a common and dangerous mistake developers make when configuring the AWS cloud.

Permissions can be collected into policies. A policy consists of one or more permissions

that are somehow related. You can then attach policies to users. A policy named `staging-app-dev` can allow to manage buckets for the staging application but not other buckets in the AWS account.

Configuration

In the next section, we will write a code that will upload file to our bucket. To make it happen, we need to create a policy that includes this permission and then attach it to our user.

Navigate to the **IAM** service and click on the **Policies** link on the list on the left side of the screen. You will see a list of policies. AWS manages those policies; they are predefined, and you cannot change them, but you can use them for common operations instead of defining your own:

The screenshot shows the AWS IAM console 'Policies' page. At the top, there's a breadcrumb 'IAM > Policies' and a header 'Policies (1166) Info'. Below the header, there's a description: 'A policy is an object in AWS that defines permissions.' To the right of the description are buttons for 'Actions', 'Delete', and 'Create policy'. A search bar contains 's3' and a 'Filter by Type' dropdown is set to 'All types', showing '12 matches'. Below this is a table of policies:

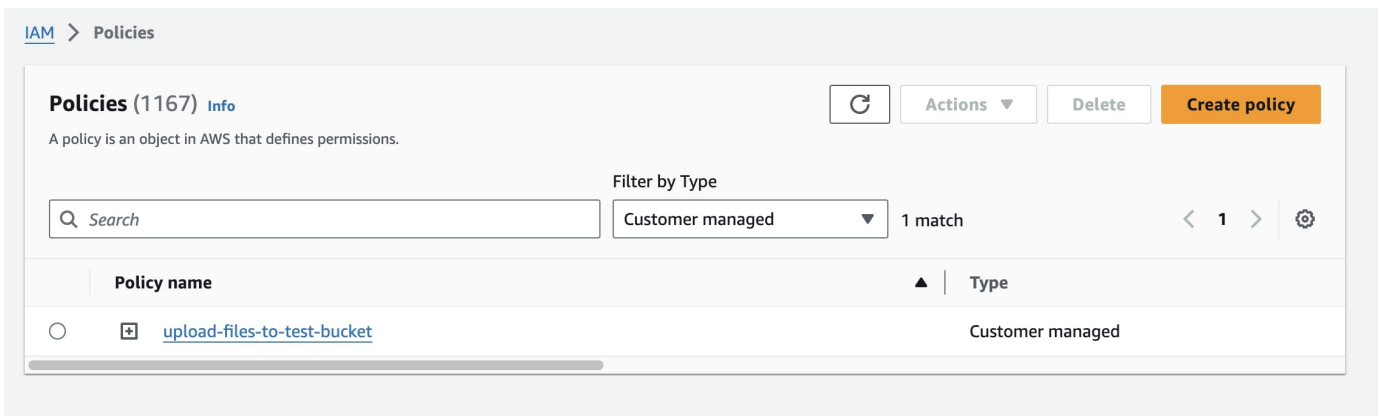
Policy name	Type
AmazonDMSRedshiftS3Role	AWS managed
AmazonS3FullAccess	AWS managed
AmazonS3ObjectLambdaExecutionRolePolicy	AWS managed
AmazonS3OutpostsFullAccess	AWS managed
AmazonS3OutpostsReadOnlyAccess	AWS managed
AmazonS3ReadOnlyAccess	AWS managed
AWSBackupServiceRolePolicyForS3Backup	AWS managed
AWSBackupServiceRolePolicyForS3Restore	AWS managed
AWSS3OnOutpostsServiceRolePolicy	AWS managed
IVSRecordToS3	AWS managed
QuickSightAccessForS3StorageManagementAnalyticsReadOnly	AWS managed
S3StorageLensServiceRolePolicy	AWS managed

Since we want to allow only upload files, the policy `AmazonS3FullAccess` gives too much space. We need to create our own; let's do it now. Click on the `Create policy`

button, and this time, we are going to use policy creator instead of raw JSON:

- For the service, select s3
- Toggle the "write" list and select the "PutObject" action
- In the "Resources" section, click "Add ARNs" - ARN is the unique identifier; in our case, the bucket identifier
- In the ARN modal, type the bucket name and click on the "Any object name" bucket - confirm your choice
- Click "Next" and provide the policy name. Try to add a meaningful name that describes the set of permissions. I named my policy `upload-files-to-test-bucket`
- Click "Create policy"

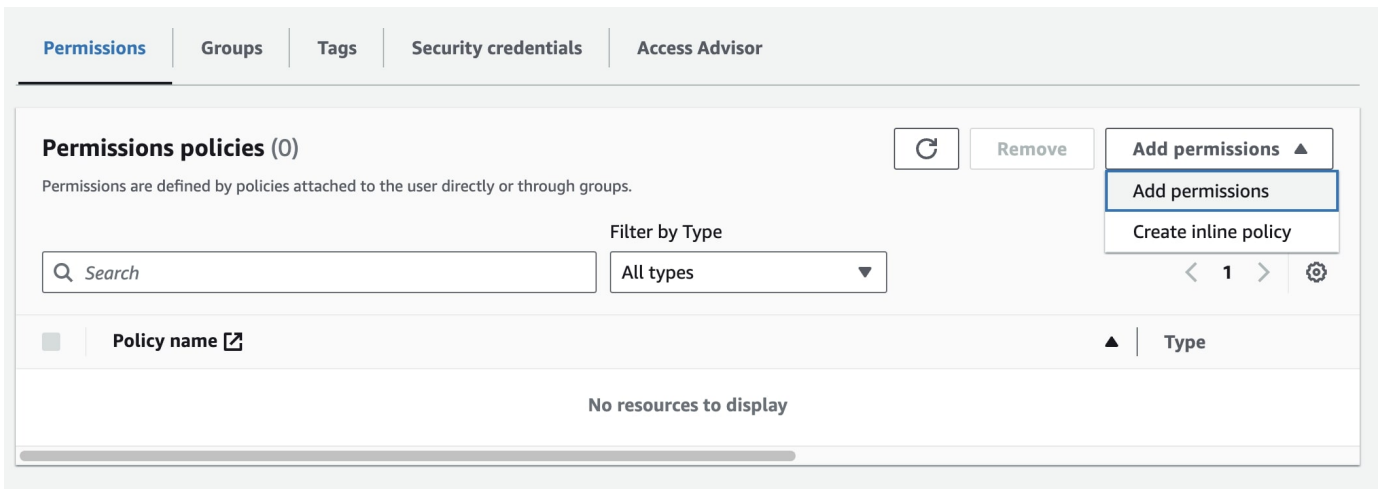
You should be able to see your new policy on the list when filtering by "customer managed" policy type:



Attaching policies to users

The last step is to assign a policy to the given user so the API keys that we have for this user will allow us to upload files to the bucket. Navigate to **IAM** service, select the **Users** link on the list on the right, and click on your user.

You can now scroll down to the permissions section and click on the "Add permissions" button:



Select "Attach policies directly", find the policy you created a while ago, mark it, and click "Next". Click "Add policies" on the confirmation screen and the process is finished.

If you manage more users, a good practice is to create roles and assign policies to roles instead of users.

The configuration phase is over, and we can finally start writing some Ruby code to interact with the AWS services.

Development

If you haven't created a sample Rails application with support for ENV variables yet, go back to the introduction chapter, where I demonstrated how to do it quickly.

In terms of AWS, we are not going to use a raw API. Amazon provides SDK for Ruby, which works well, so there is no need to reinvent the wheel. In this section, I will review a few real-world examples of features you can create with S3 and Rails. First, let's configure and establish the connection with S3.

Connection configuration

Enter the project directory and install the s3 SDK:

```
bundle add aws-sdk-s3
```

I'm going to organize the code related to S3 in service called `app/services/aws_s3.rb`, and here is the initial code that we will extend in a minute:

```

class AwsS3
  private

  def client
    @client ||= Aws::S3::Client.new(
      access_key_id: ENV.fetch('AWS_ACCESS_KEY_ID'),
      secret_access_key: ENV.fetch('AWS_SECRET_ACCESS_KEY'),
      region: ENV.fetch('AWS_REGION')
    )
  end
end

```

We are now ready to implement the first feature - uploading files to the test bucket.

Uploading single files to the bucket

As you may remember, the action related to uploading files to the bucket was named "PutObject". The method named the same way is available in the SDK and requires the bucket name and key. Additionally, we will provide the file content:

```

class AwsS3
  def upload_file(bucket:, key:, body:)
    client.put_object(bucket: bucket, key: key, body: body)
  end

  private

  def client
    @client ||= Aws::S3::Client.new(
      access_key_id: ENV.fetch('AWS_ACCESS_KEY_ID'),
      secret_access_key: ENV.fetch('AWS_SECRET_ACCESS_KEY'),
      region: ENV.fetch('AWS_REGION')
    )
  end
end

```

Let's create the file we will upload to the bucket:

```
echo "Ruby" >> ./tmp/best-language.txt
```

Open the Rails console and upload the file using our service:

```
service = AwsS3.new
service.upload_file(bucket: 'bucket-name', key: 'best-language.txt',
body: File.read('./tmp/best-language.txt'))
```

Replace `bucket-name` with your bucket name. A good practice is to fetch a bucket name with a meaningful name from the environment variable. Here, I made it quicker by hardcoding the value.

Go to the AWS account, open the test bucket, and verify if the file was uploaded correctly.

Reading files from the S3 bucket

Since we uploaded a text file that contains the name of the best programming language, let's pull it from the bucket and see what the name of the technology is:

```
class AwsS3
  # ...

  def get_file(bucket:, key:)
    client.get_object(bucket: bucket, key: key)
  end

  # ...
end
```

Like before, let's open the console and call the service:

```
service = AwsS3.new
service.get_file(bucket: 'bucket-name', key: 'best-language.txt')
```

Instead of the name of the best programming language, we've got the following error: `Aws::S3::Errors::AccessDenied`. It happened because the policy attached to our AWS keys allows only for file upload, not reading files that exist in the bucket.

We can update our existing policy, but since we can't update the name of the policy, it's better to delete this policy and add a new one with a more meaningful name - `manage-test-bucket`.

Do you know what change we need to make in the policy definition? It's `s3:GetObject`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::railsonaws/*"
    }
  ]
}
```

Go ahead and delete the old policy, create a new policy, and assign it to our user. When it's done, restart the console and rerun the code:

```
service = AwsS3.new
file = service.get_file(bucket: 'bucket-name', key: 'best-language.txt')
file.body.read # => "Ruby"
```


As I thought Ruby is the best language. But there is a problem. We need to tell the world about it, but we can't because the file is not publicly available.

Generating presigned URLs

We can make the bucket files public by default, but we want to share only this file. Generating unique and expiring links to files is a valuable feature widely used in Rails applications.

Let's modify our service and provide a method for generating a presigned URL:

```
class AwsS3
  # ...

  def generate_presigned_url(bucket:, key:, expires_in:)
    signer = Aws::S3::Presigner.new(client: client)
    signer.presigned_url(:get_object, bucket: bucket, key: key,
expires_in: expires_in.to_i)
  end

  # ...
end
```

This time, we don't need any additional permissions; we can go ahead and generate the URL to our file:

```
service = AwsS3.new
service.generate_presigned_url(bucket: 'railsonaws', key: 'best-
language.txt', expires_in: 2.minutes)
```

Listing files in the bucket

Another standard action performed with S3 in Rails applications is getting the names of all files in the given bucket. This time, we have to update our policy with another permission that is not related to all files but to the bucket itself:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::bucket-name/*"
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": "arn:aws:s3:::bucket-name"
    }
  ]
}

```

The above policy is a good example of multiple permissions in one policy. We can now update our service:

```

class AwsS3
  # ...

  def files_in_bucket(bucket)
    client.list_objects_v2(bucket: bucket).contents.map(&:key)
  end

  # ...

```

end

We can do our test to confirm that everything is working correctly. The above code looks simple, but it won't work if more than 1000 files are in the bucket. For listing larger buckets, we need to implement pagination:

```
def files_in_bucket(bucket:, limit: 1_000)
  initial_response = client.list_objects_v2(bucket: bucket, max_keys:
limit)
  files = initial_response.contents.map(&:key)
  next_continuation_token = initial_response.next_continuation_token

  while next_continuation_token.present?
    response = client.list_objects_v2(bucket: bucket, max_keys: limit,
continuation_token: next_continuation_token)
    files += response.contents.map(&:key)
    next_continuation_token = response.next_continuation_token
  end

  files
end
```

Using s3 to store attachments from ActiveRecord models

The newer version of Rails comes with the Active Storage library, allowing us to handle attachments in the application easily. The good news is that the library allows for different storage facilities, including s3. Let's see how we can configure our application to store all files in one of the buckets.

Active Storage installation

The installation process consists of 2 simple steps. We have to trigger the installation command, which will generate migration, and then we have to run the migration:

```
rails active_storage:install
./bin/rails db:migrate
```

Active storage is ready to use, but we don't have any models yet to which we can attach any files. Let's change that situation.

Sample model generation

I will make it simple. Let's generate a `User` model with the `name` column, and a single user will have one avatar:

```
./bin/rails g model User name:string avatar:attachment
./bin/rails db:migrate
```

You can now inspect the `User` model, and you will notice that it contains the `has_one_attached :avatar` instruction, which tells Active Storage that we would like to attach one file to each record, and it would be named `avatar`.

S3 driver configuration

The configuration of different drivers for Active Storage is placed in the `config/storage.yml` file; let's uncomment the `amazon` entry and update credentials:

```
amazon:
  service: S3
  access_key_id: <%= ENV['AWS_ACCESS_KEY_ID'] %>
  secret_access_key: <%= ENV['AWS_SECRET_ACCESS_KEY'] %>
  region: <%= ENV['AWS_REGION'] %>
  bucket: bucket-name
```

The Amazon driver is ready, but we need to tell Rails that we want to store files on `s3`, not on the disk for the development environment. We need to update the `config/environments/development.rb` file with the following entry:

```
config.active_storage.service = :amazon
```

Also, make sure that for the configured keys you assigned the following policy that contains all actions required by Active Storage to work correctly:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:DeleteObject",
        "s3:PutObjectAcl"
      ],
      "Resource": "arn:aws:s3:::bucket-name/*"
    },
    {
      "Effect": "Allow",
      "Action": "s3:ListBucket",
      "Resource": [
        "arn:aws:s3:::bucket-name"
      ]
    }
  ]
}
```

Verifying attachment upload

You will need a simple image of any format or size to test the code. I downloaded a simple user avatar in PNG format. Let's open the Rails console and test:

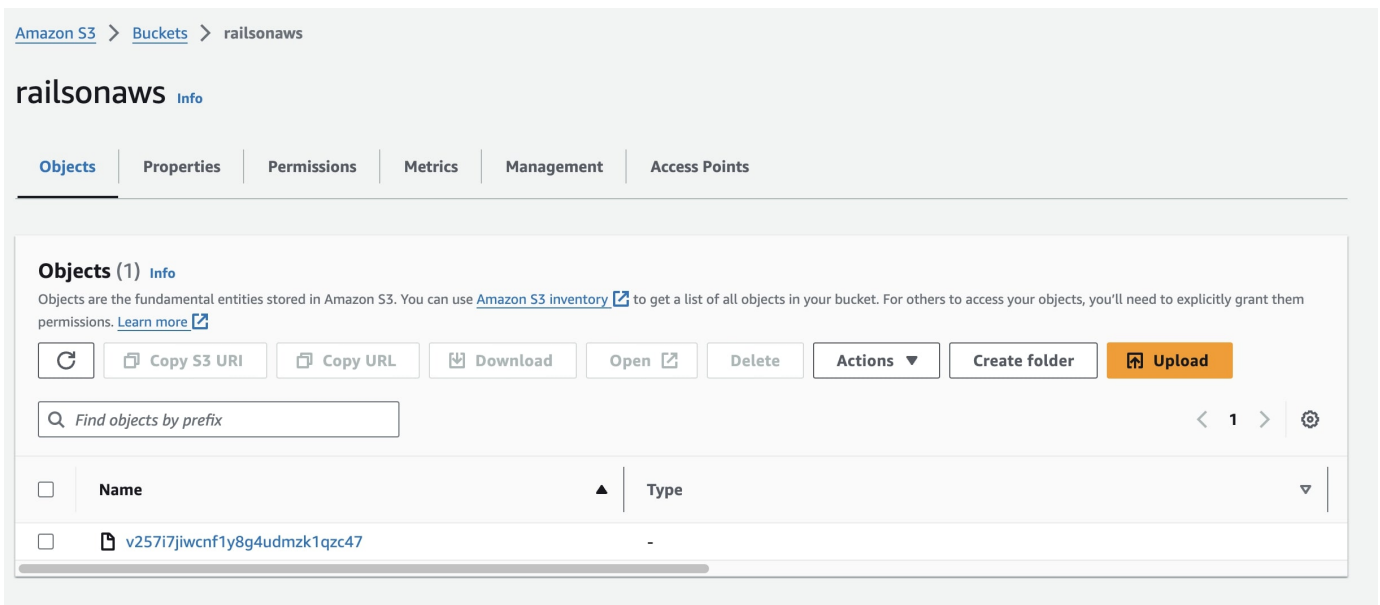
```
user = User.create!(name: 'John Doe')
```

```
user.avatar.attached? # => false
```

```
user.avatar.attach(File.open('./avatar.png'))
```

```
user.avatar.attached? # => true
```

You can also take a look at the bucket. You should see there is a new file with a strange tokenized name:



We are sure the file was uploaded, but let's test if we can display it in our application.

Rendering images from s3

For the test, I will create a simple controller, assign the first user, and try to render the avatar in the view. Create the HomeController:

```
class HomeController < ApplicationController
  def index
    @user = User.find_by!(name: 'John Doe')
  end
end
```

Define a simple view in `app/views/home/index.html.erb`:

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

The last step is to update the `config/routes.rb` file to render the view as the root page of our application:

```
Rails.application.routes.draw do
  root "home#index"
end
```

Run the rails server with the `rails s` command and visit localhost to verify that the avatar is displayed correctly. If you look into the console logs, you will notice that Rails created a presigned URL to render the image. Refresh the page, and you will see that the new link has not been generated, as the old one hasn't expired yet.

