ruby on

Rails on AWS

cloud

Architecture, networking, security, and pricing

2ND EDITION

Paweł Dąbrowski

Development Driven Publishing

iRonin.IT

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:

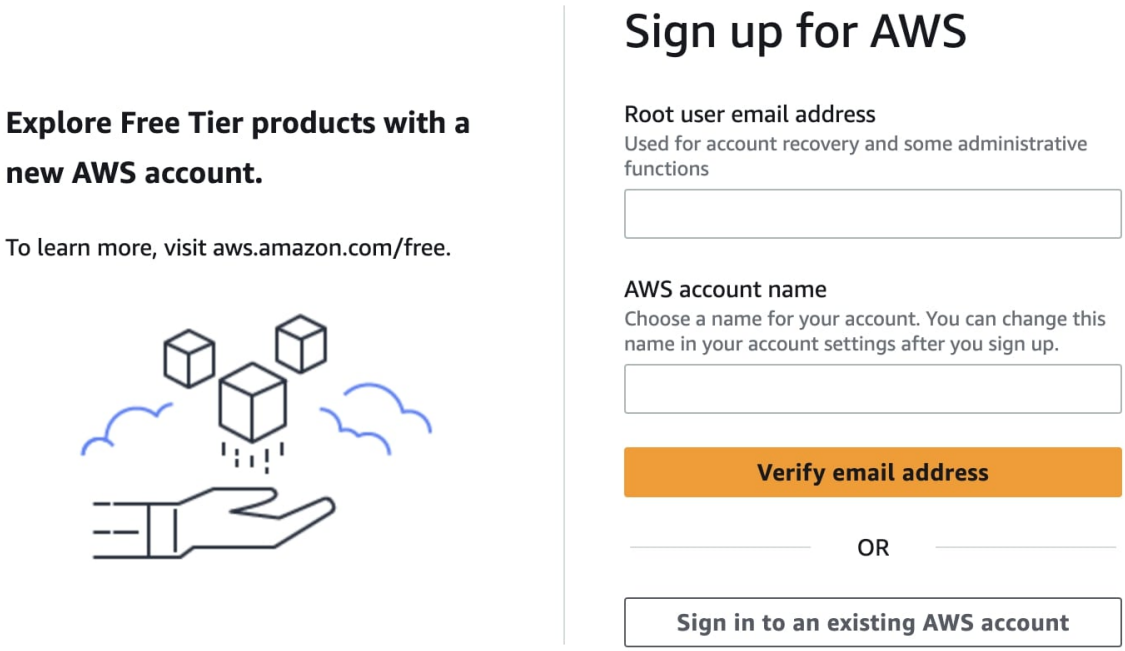
- **Basic Rails application** - Ruby 3.2.2 and Rails 7.1.2
- **AWS account** - you can set up it for free and then use the free tier of services not to produce any costs

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:



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

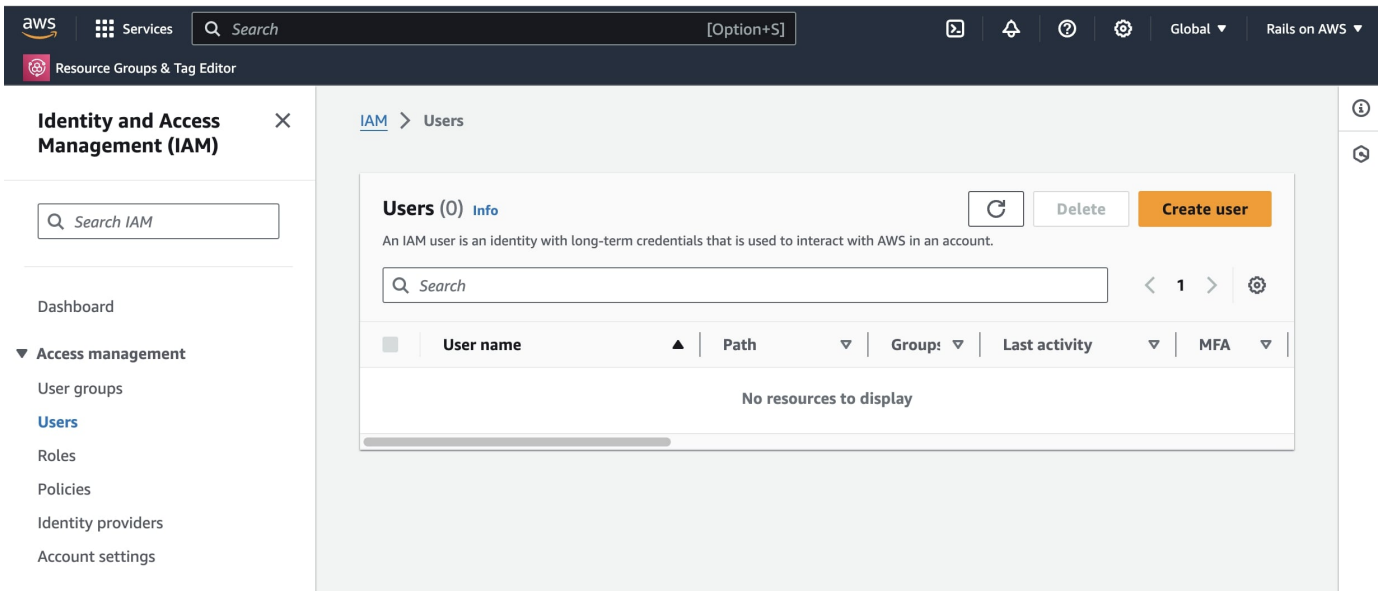
Each user that is using a given AWS account should have separate credentials. You can access all services on the platform as a root account, but we will create a separate account to show you how to add permissions securely.

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.

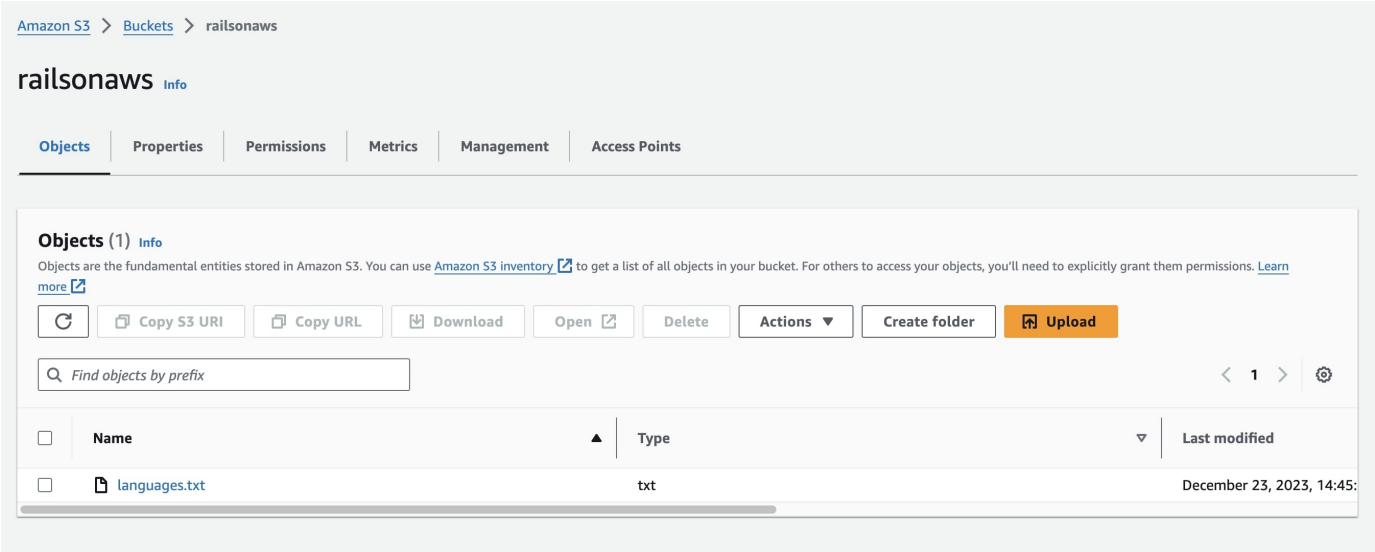
Explanation

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

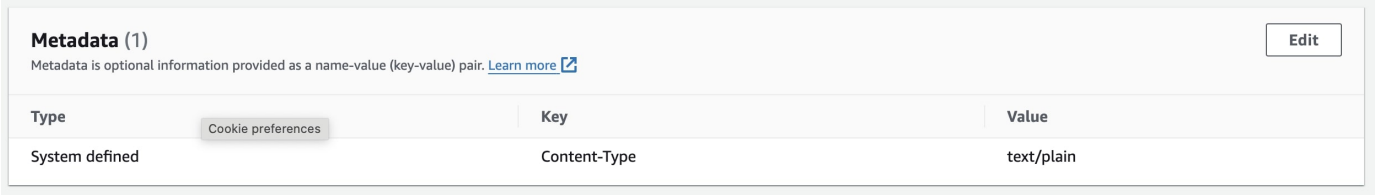
You won't be able to access https://railsonaws.s3.eu-central-1.amazonaws.com/text and get a tree of files inside the directory as it would be possible in Linux, Windows, or Mac OS.

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:



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.

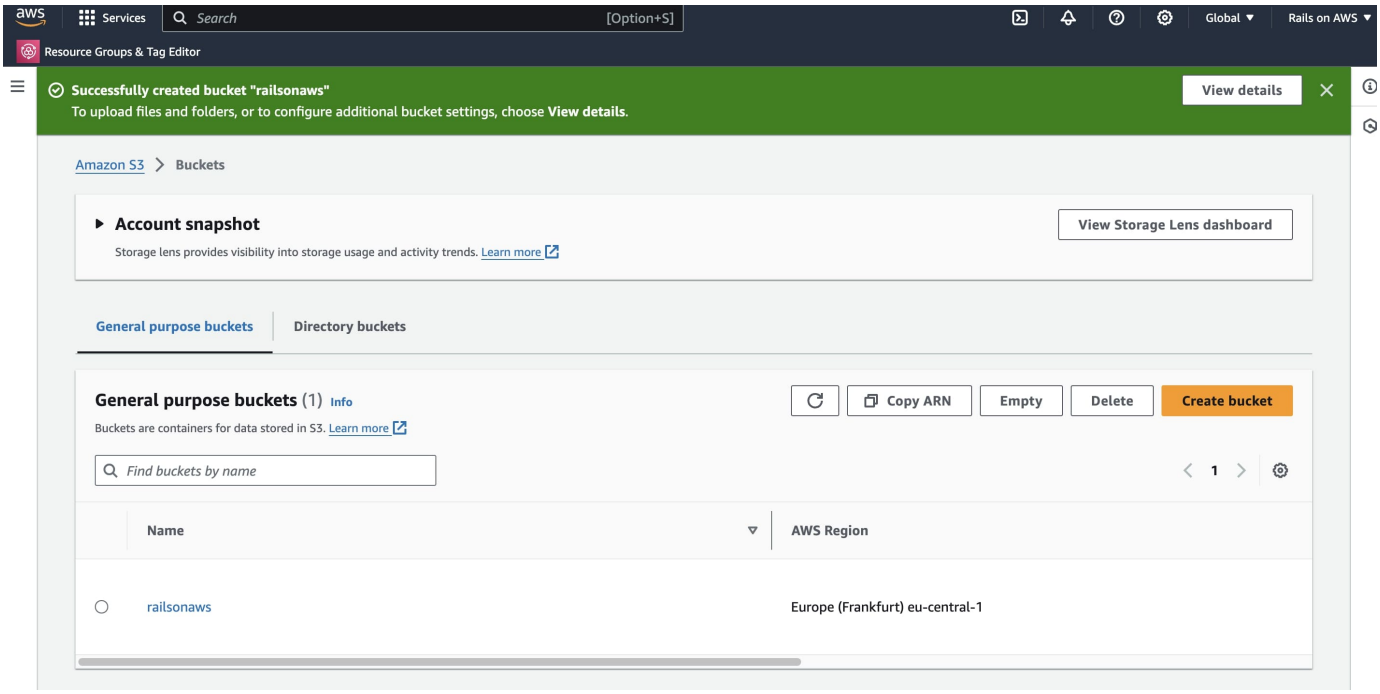
Of course, there are many more use cases. One of the most significant advantages of using S3 over standard server storage is that in the containerized application, you wouldn't be able to access files generated by other application instance without uploading them to the cloud.

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:



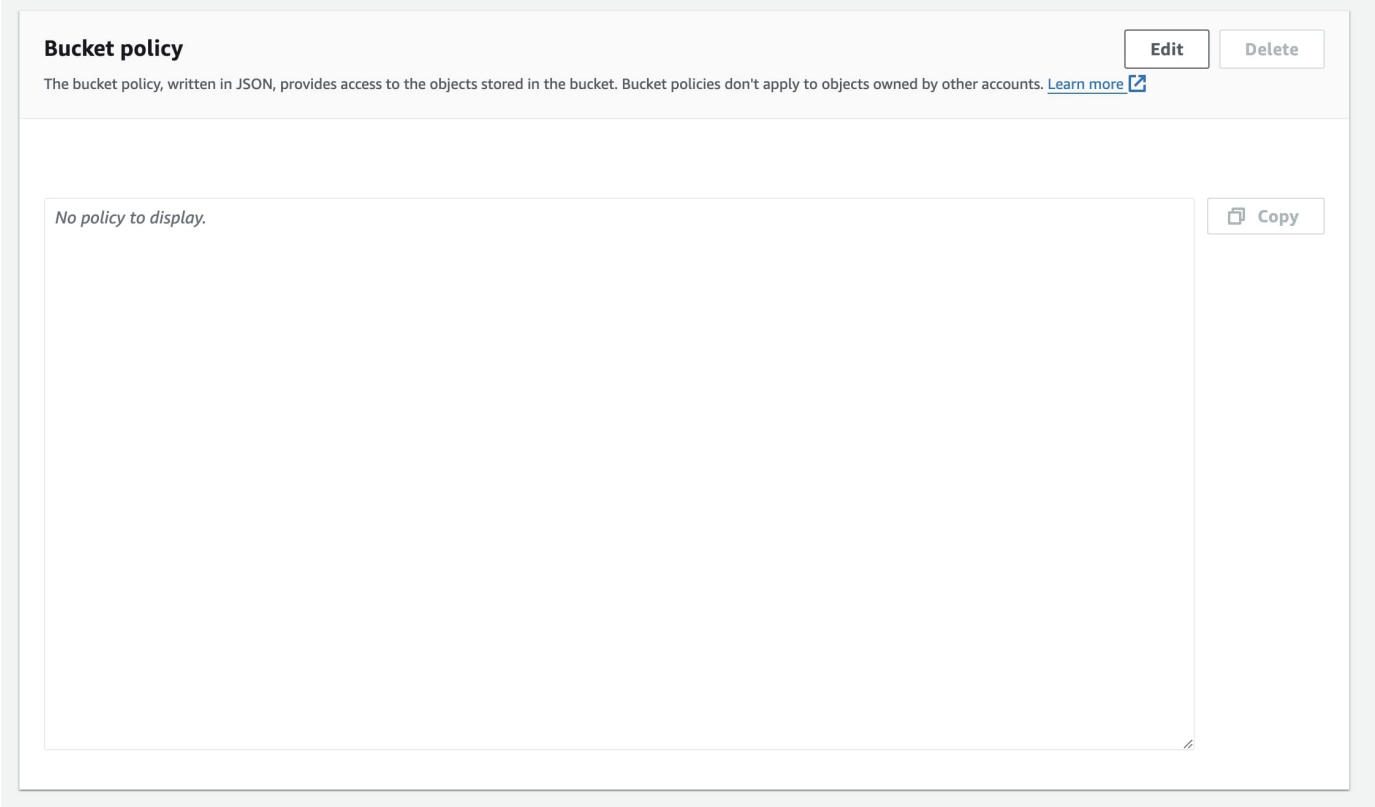
Files upload

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/*"
      ]
    }
  ]
}
```

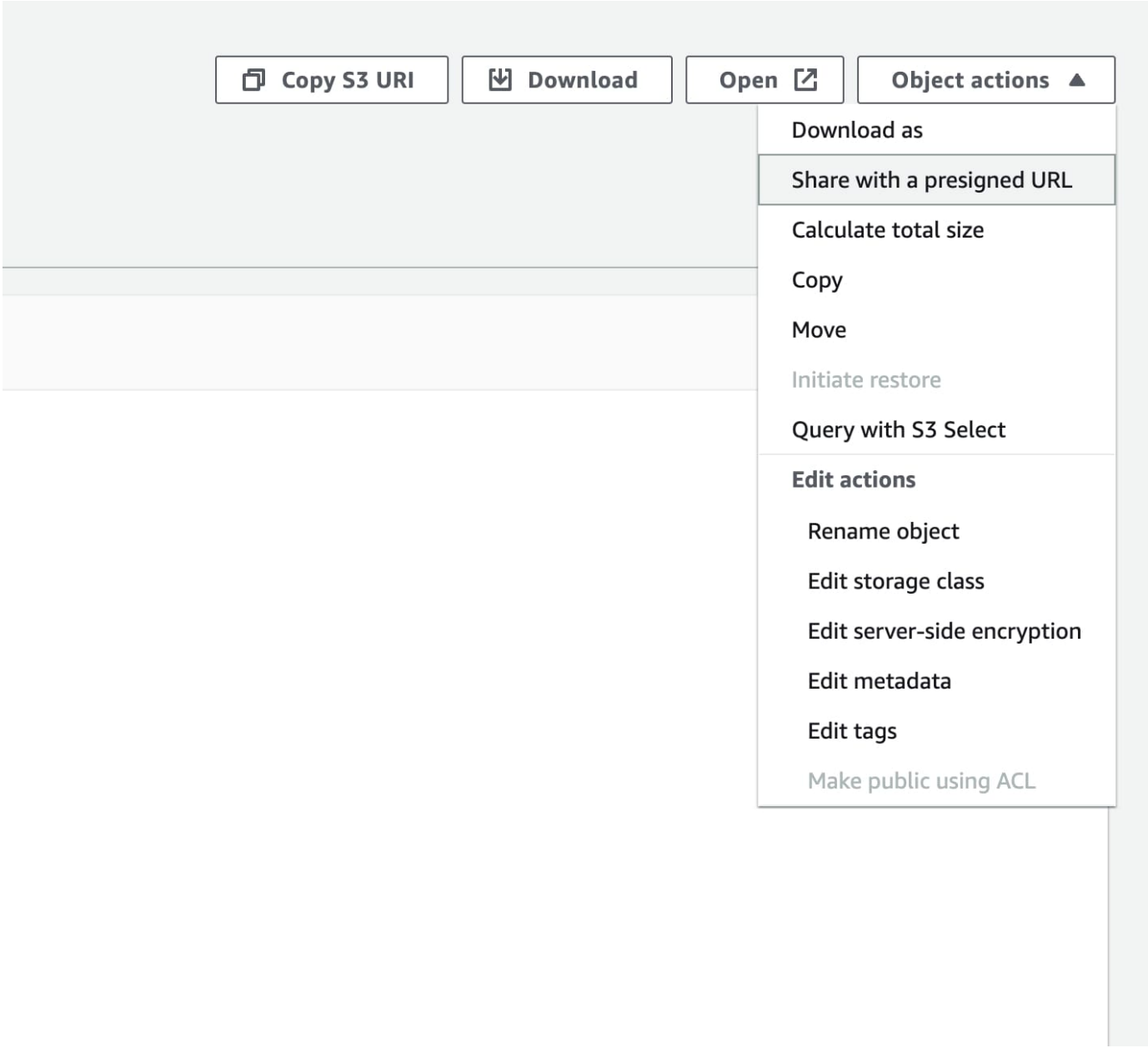
In a shortcut, we allow (Effect attribute) anyone (Principal attribute) to perform action s3:GetObject (show the file) in resource arn:aws:s3:::bucket-name/* - replace bucket-name with the name of your bucket. Paste the policy content with the changed bucket name and save the changes. Now refresh the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

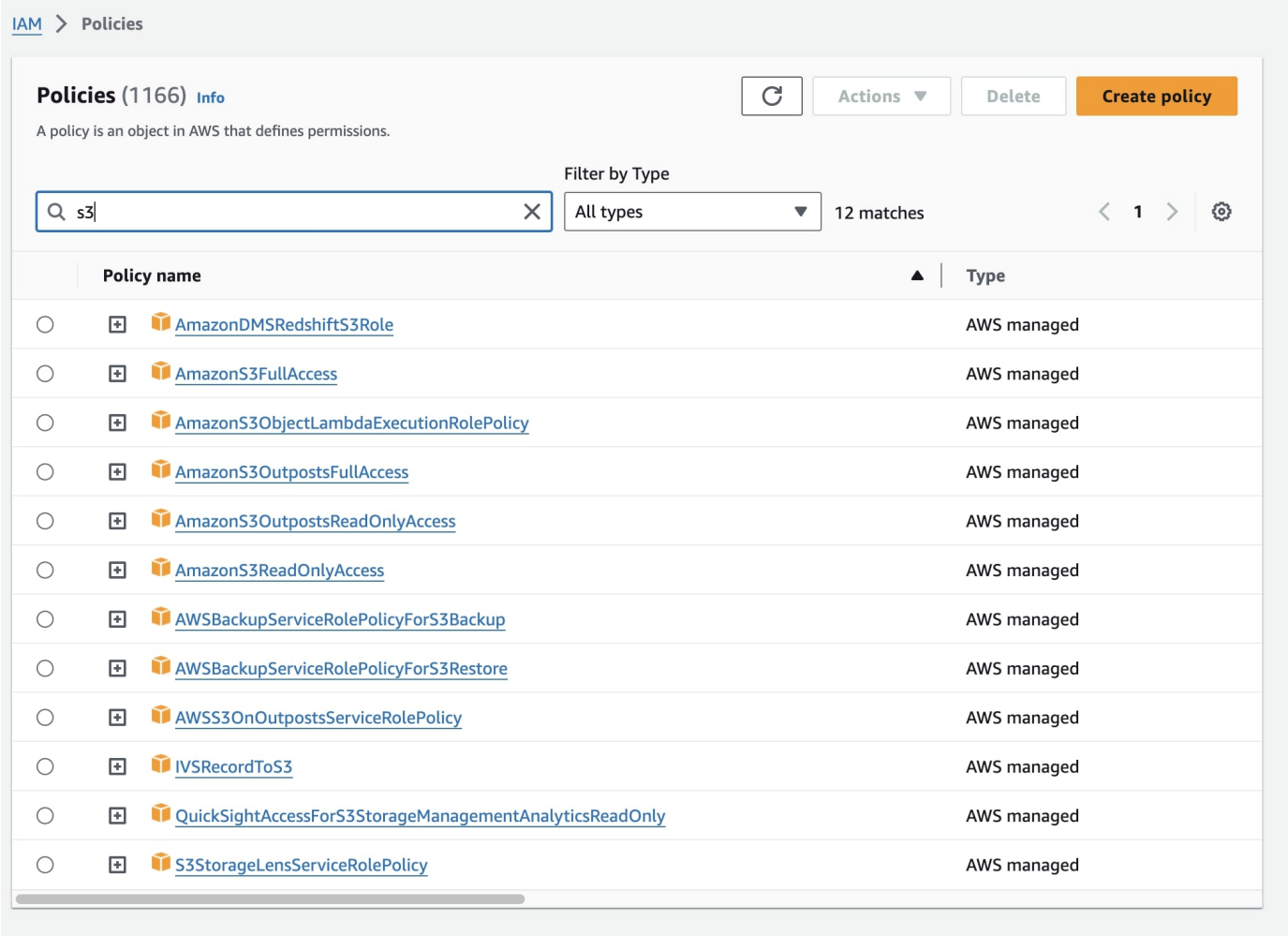
Permissions can be collected into policies. A policy consists of one or more permissions

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:

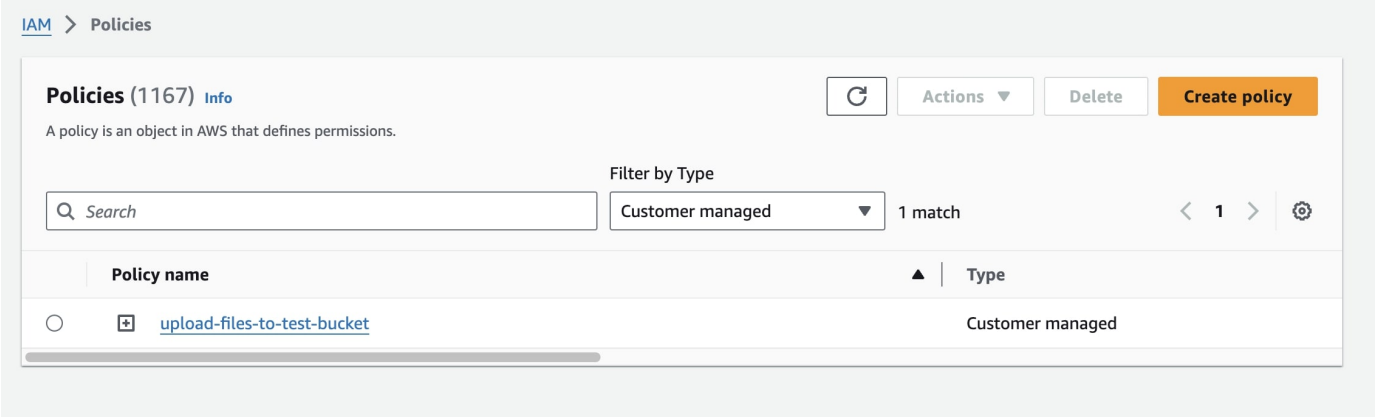


Since we want to allow only upload files, the policy AmazonS3FullAccess gives too much space. We need to create our own; let's do it now. Click on the Create policy

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"

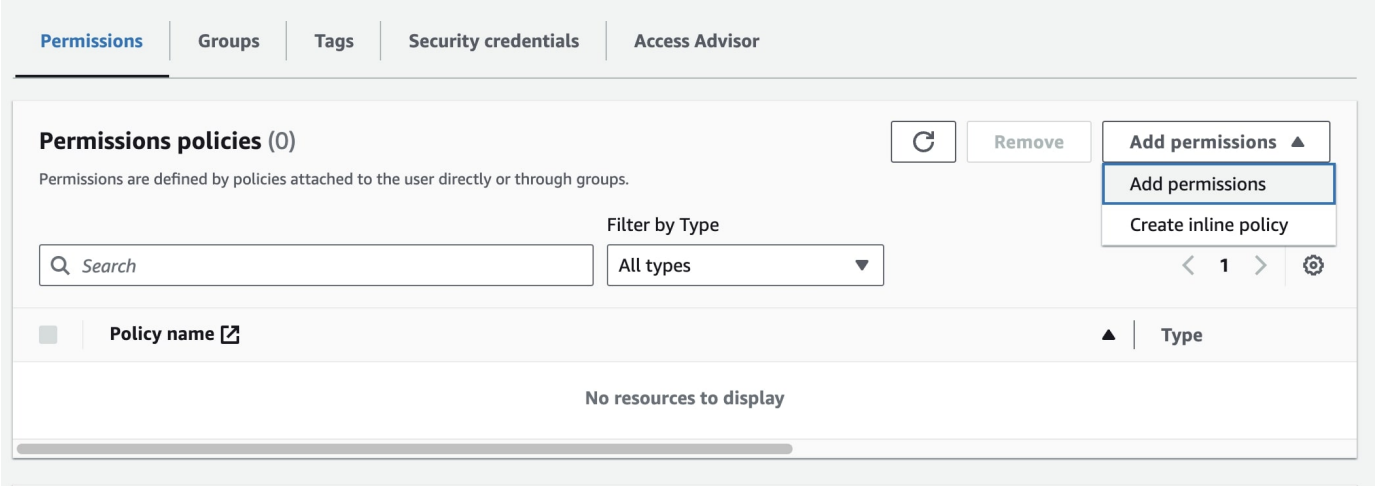
You should be able to see your new policy on the list when filtering by "customer managed" policy type:



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"
      ]
    }
  ]
}
```

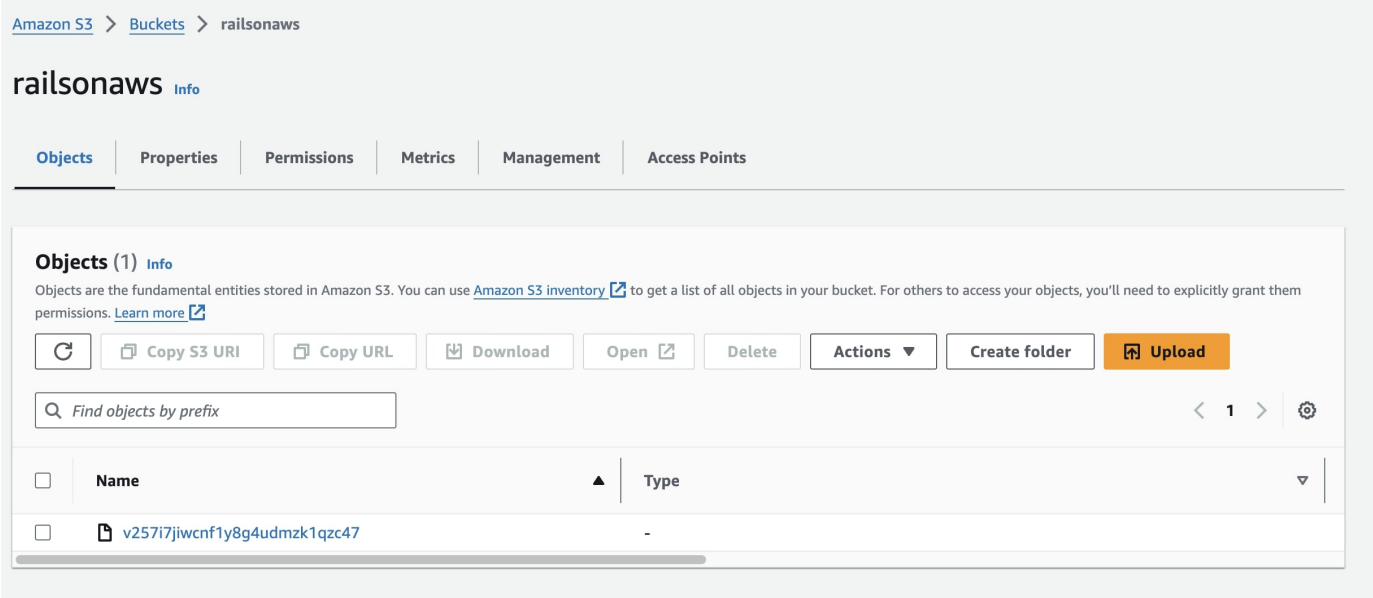
Verifying attachment upload

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:

```ruby
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:

```ruby
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.

CloudFront

Cloudfront is Amazon’s Content Delivery Network (CDN) solution. As a shortcut, CDN is a network of servers located worldwide that serves your website's assets mainly to improve its performance. Imagine that you host your servers in Europe, and some visitors come from the United States. It will take some time to load images or other assets. Thanks to CDN, your assets are cached in US-based servers, so loading the files is much shorter.

In this chapter, you will learn the minimum required to manage Cloudfront 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
- **Pricing** - I will discuss the general pricing for the service
- **Configuration** - I will walk you through the configuration process of service to make it ready for your Rails application
- **Development** - we will create code together to show you how to utilize Cloudfront service in your Rails application

AWS provides a free tier for Cloudfront service, which is more than enough for tests. I will discuss the details of the free tier in the pricing section.

Explanation

As I mentioned before, CDN is a network of servers. To configure it for your website, you have to create a **distribution**. You can treat one distribution as one configuration set. In most cases, you will need only one distribution per Rails application.

Distribution URL

When configuring distribution, you can set up your domain or subdomain that CloudFront will use. A lot of companies are using the assets.domain.com pattern. If you don’t define your domain / subdomain, you can use the default one provided by AWS that looks like distribution-id.cloudfront.net

Assets caching process

Imagine that you have a logo.jpg file in your application. Without the CDN, you probably would access the file with a URL similar to https://domain.com/assets/logo-67f9c4ad2cf2.jpg. With CloudFront as CDN, the process is the following:

- When the asset is requested for the first time from the given location, CloudFront calls the original server address, caches the file, and returns the asset to the user
- When the asset is requested not for the first time from the given location, the cached version of the asset is returned to the user

CloudFront automatically selects the server that is closest to the visitor. Those servers that keep cached assets are called **edge locations**. AWS located edge locations around the world.

Cache expirations

By default, CloudFront will remove the cache from the servers after 24 hours. Of course, you can change this behavior to adjust to the nature of the assets that you serve.

You should expire the cache quicker if you serve more dynamic content. If your files don't change often, you might want to expire after longer periods. You can change the expiration of the cache in the following ways:

- You can update CloudFront settings to adjust the cache duration for all files that match the same path pattern
- You can modify the cache duration for individual file with the Cache-Control header
- You can also force certain assets to be removed from the cache; the process is named **invalidation**. You can invalidate single files or all files that match the same path pattern

Later in this chapter, I will show you how to update expiration settings for multiple or single files by updating CloudFront configuration or performing invalidation from the AWS dashboard.

Available configuration settings

When you create CloudFront distribution, you have a few settings to adjust the CDN to your application's needs. Available are the following:

- **Content origin** - the place from where the CDN will get the files that are later cached. In our case, it will be our Rails application server. Besides any HTTP server, you can also use some AWS services like S3, Elastic Load Balancer, and more.
- **Access** - you can decide whether the files should be available to everyone or restricted to some users.
- **Security** - you can decide to force your users to use HTTPS or enable AWS WAF protection
- **Cache key** - uniquely identifies each file in the cache, and you can decide which values to include in this key
- **Origin request settings** - you can decide if CloudFront should include HTTP headers, cookies, or query strings in requests that it sends to your server
- **Geographic restrictions** - if you don't want users in certain countries to access your content, you can achieve this by using this setting
- **Logs** - you can collect standard logs or real-time logs to review the viewer user activity

Set up a few distributions with different configuration settings if you have many use cases for a single application.

Pricing

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

- **Data transfer out** - you pay for the gigabytes of data fetched from the AWS cache (stored on edge locations). The more data is fetched, the less you spend, and the exact price per GB differs for various regions. For example, for Europe, you will pay $0.085 per GB for the first 10 TB of data, and for Japan, you will pay $0.114.

- **HTTP or HTTPS requests** - you pay for each request performed to the CloudFront distribution. You always pay the same price regarding requests, which won't be lower if you perform more. The pricing is set per 10,000 requests, and it's a little bit different for different locations

Besides the base functionality of CDN, you can also use viewer functions, which are great if you want to manipulate cache keys or URL rewrites, but it's beyond the scope of this book. Viewer functions have separate pricing.

Free tier

As I mentioned at the beginning, AWS provides a free tier for this service, so you can play with it (or even use it for your production application to some extent) without the need to pay for anything.

You won't pay a cent per month for using CloudFront if you don't exceed the following limits:

- 1 TB of data transferred out to the internet
- 10,000,000 HTTP or HTTPS requests
- 2,000,000 CloudFront Function invocations
- 2,000,000 CloudFront KeyValueStore reads

I haven't mentioned anything about the KeyValyeStore yet. It's a global key-value datastore that you can use within CloudFront Functions. I won't be using it in this book.

Configuration and development

Let's create a simple Rails application that renders only a few images. I will configure CloudFront distribution to proxy asset access to provide better performance. I will also demonstrate how to invalidate a given asset if we want it to be purged from the cache before the automatic purge process.

Rails application generation

This time, we do not need to use any database:

```
rails _7.1.2_ new cloudfront
cd cloudfront/
```

We need one default controller with a view. Start with creating app/controllers/home_controller.rb:

```
class HomeController < ApplicationController
  def index; end
end
```

And with a blank view placed at /app/views/home/index.html.erb. The last step is to update routes:

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

Rendering sample images

We will need some images to demonstrate how the CDN is working. I'm going to visit https://pexels.com to find some images of Alaskan Malamutes. Go ahead and select three images, download them, and put them into the app/assets/images directory in your Rails application.

Name the files image1.jpg, image2.jpg, and image3.jpg so it will be easier to follow the next instructions. Update app/views/home/index.html.erb view to render images:

```
<div>
  <%= image_tag('image1.jpg', style: 'max-width: 300px;') %>
</div>
<div>
  <%= image_tag('image2.jpg', style: 'max-width: 300px;') %>
</div>
```

```
<div>
  <%= image_tag('image3.jpg', style: 'max-width: 300px;') %>
</div>
```

Configuring CORS

Because we are going to render images via CDN, we need to set the proper CORS settings in Rails app. Start with installing the gem:

```
bundle add rack-cors
```

Then create an initializer with the following code:

```
Rails.application.config.middleware.insert_before 0, Rack::Cors do
  allow do
    origins '*'
    resource '*', headers: :any, methods: [:get, :post]
  end
end if Rails.env.production?
```

Application deployment

Our application needs to be available online to use CloudFront. You can either deploy the application to Heroku or another server provider. I will use Heroku. If you are also going to deploy the app on Heroku, here is the set of default environment variables to set:

- RAILS_ENV=production
- RACK_ENV=production
- RAILS_LOG_TO_STDOUT=enabled

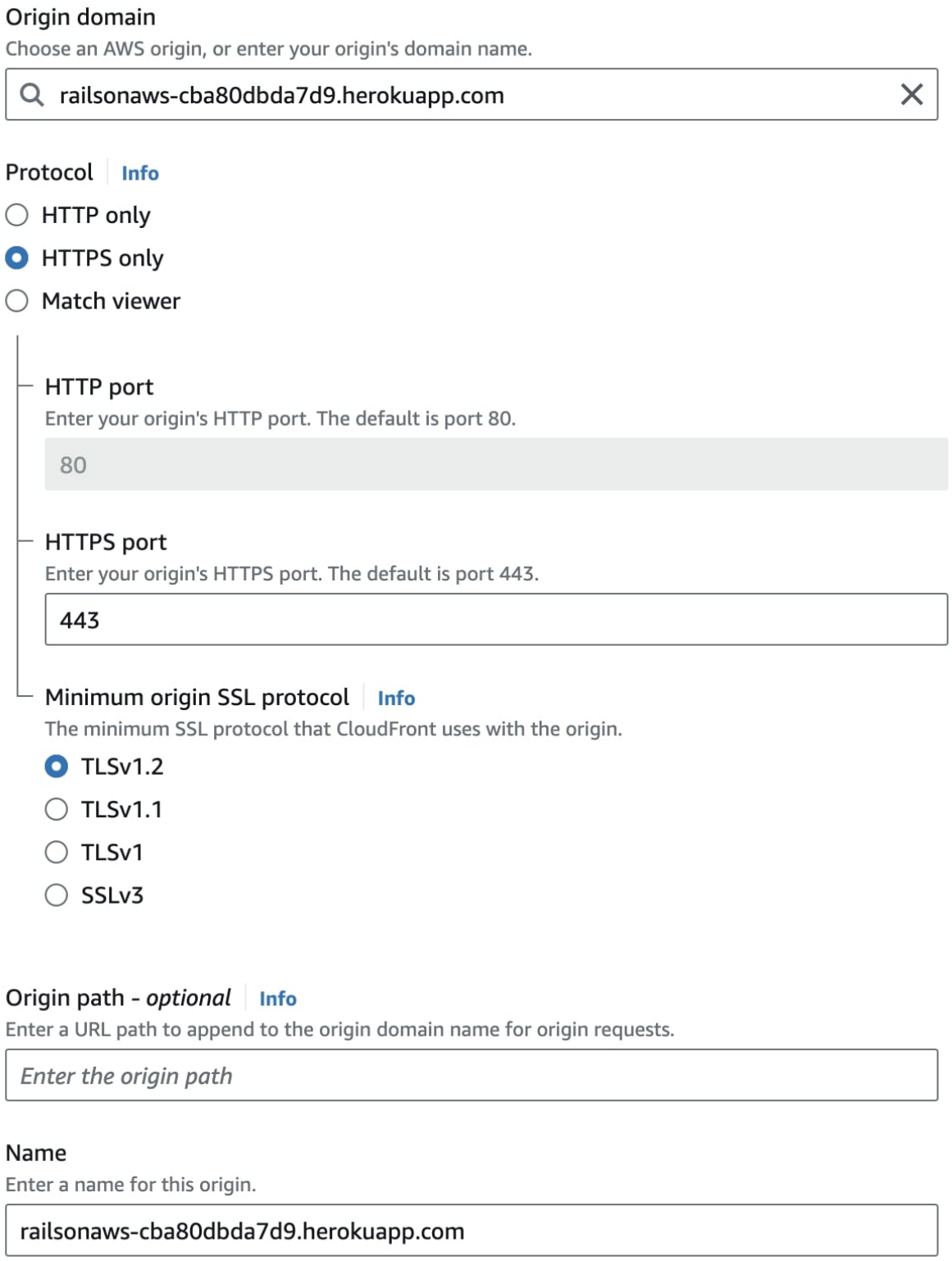
- RAILS_SERVE_STATIC_FILES=enabled
- SECRET_KEY_BASE=unique_token

You can generate the unique token for SECRET_KEY_BASE by running bundle exec rails secret. If you are going to run app on Heroku, create also Procfile in the main app directory with the following content:

```
web: bundle exec rails server -p $PORT -e $RAILS_ENV
```

CloudFront configuration

We can finally configure the first CloudFront distribution. Sign in to the AWS account, select CloudFront service, and click “Create a CloudFront distribution”.



The Origin domain should contain the address of your application. I also selected HTTPS only for protocol.

Path pattern | Info

Default (*)

Compress objects automatically | Info

- ○ No
- ◉ Yes

Viewer

Viewer protocol policy

- ○ HTTP and HTTPS
- ◉ Redirect HTTP to HTTPS
- ○ HTTPS only

Allowed HTTP methods

- ◉ GET, HEAD
- ○ GET, HEAD, OPTIONS
- ○ GET, HEAD, OPTIONS, PUT, POST, PATCH, DELETE

Restrict viewer access

If you restrict viewer access, viewers must use CloudFront signed URLs or signed cookies to access your content.

- ◉ No
- ○ Yes

Cache key and origin requests

We recommend using a cache policy and origin request policy to control the cache key and origin requests.

- ◉ Cache policy and origin request policy (recommended)
- ○ Legacy cache settings

In the “Default cache behavior” I just changed the protocol policy to redirect from HTTP to HTTPS and selected the “CahingOptimized” for cache policy.

Web Application Firewall (WAF) Info

- ○ Enable security protections
  Keep your application secure from the most common web threats and security vulnerabilities using AWS WAF. Blocked requests are stopped before they reach your web servers.
- ◉ Do not enable security protections
  Select this option if your application does not need security protections from AWS WAF.

The last step was to select that I didn't want to use WAF. You don't have to edit anything; you can click "Create distribution". AWS will need a few minutes to deploy the distribution.

Using CloudFront distribution

If you navigate to the main CloudFront dashboard, you will see your new distribution listed along with the URL. Mine is d2wsbra4agc1ds.cloudfront.net. In the browser, switch to your deployed application and copy the URL address of one of the images visible on the front page. For me, it's https://railsonaws-cba80dbda7d9.herokuapp.com/assets/image1-a5d951efcc01950c4ed5f473bbb88e5f348bc21d5b001316b6b418fa84173025.jpg

Now, replace your domain with the CloudFront URL and request the same path for the image. It will take some time to load the image. Now, refresh the button, and the image should appear immediately. Congratulations, you just saved the first item in the CloudFront cache.

Updating Rails to use CloudFront distribution

Our Rails application still needs to be made aware of the distribution we have just created. We don't have to type the address for the image manually; we can update the environment configuration placed in config/environments/production.rb:

```
config.asset_host = "cloudfront_url"
```

Apply the change and redeploy your application. You can now check the page's source code, and you will notice that Rails automatically applied the CloudFront distribution URL to every asset you rendered.

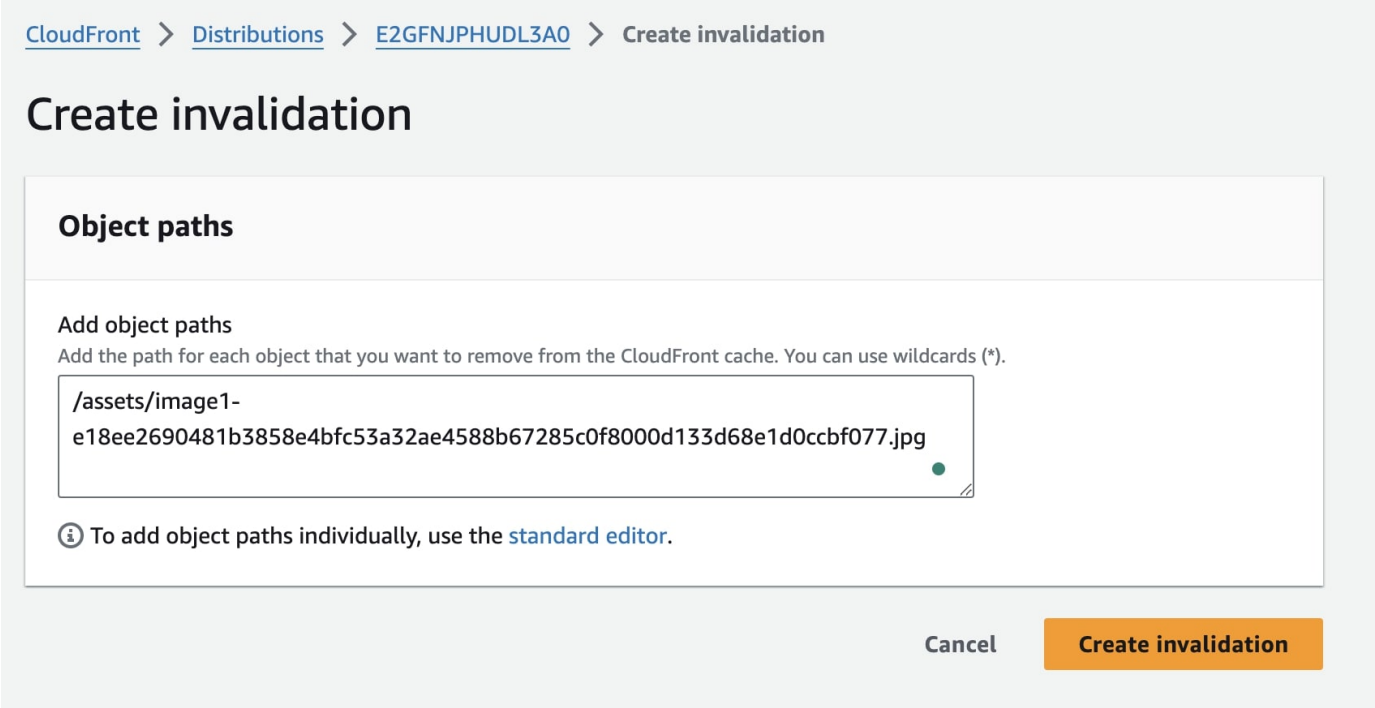
By default, the cache on CloudFront is purged every 24 hours. If you replace the image with the same name, you will notice the change in the application because there is a unique digest attached to every version of the image, preventing the caching of old photos in the browser.

Clearing the CloudFront cache manually

You can do it manually if you don't want to wait 24 hours to clear the cache. You can remove the cache for a single file, multiple paths, or all files that match the given pattern.

To do this, visit the CloudFront dashboard, click on your distribution, and then move to the "Invalidations" tab. Your list should be empty unless you performed some invalidations for this distribution in the past. Click on the "Create invalidation" button.

You can now select one image path from your application. Before clearing the cache, you can request the URL using Postman or a similar tool to see the loading time for the image. For me, it shows 72 ms. Now, input the image path to the form:



Click on "Create invalidation". It can take some time before the cache is cleared. Thankfully, there is a status field, so you will see when it's done. After it's done, request the image URL again. For me, the initial load took 295 ms.

You can also notice that when you request a file, and it's cached on CloudFront, the header X-Cache has a value Hit from cloudfront; otherwise, it has the value Miss from cloudfront.

Accessing private s3 bucket files

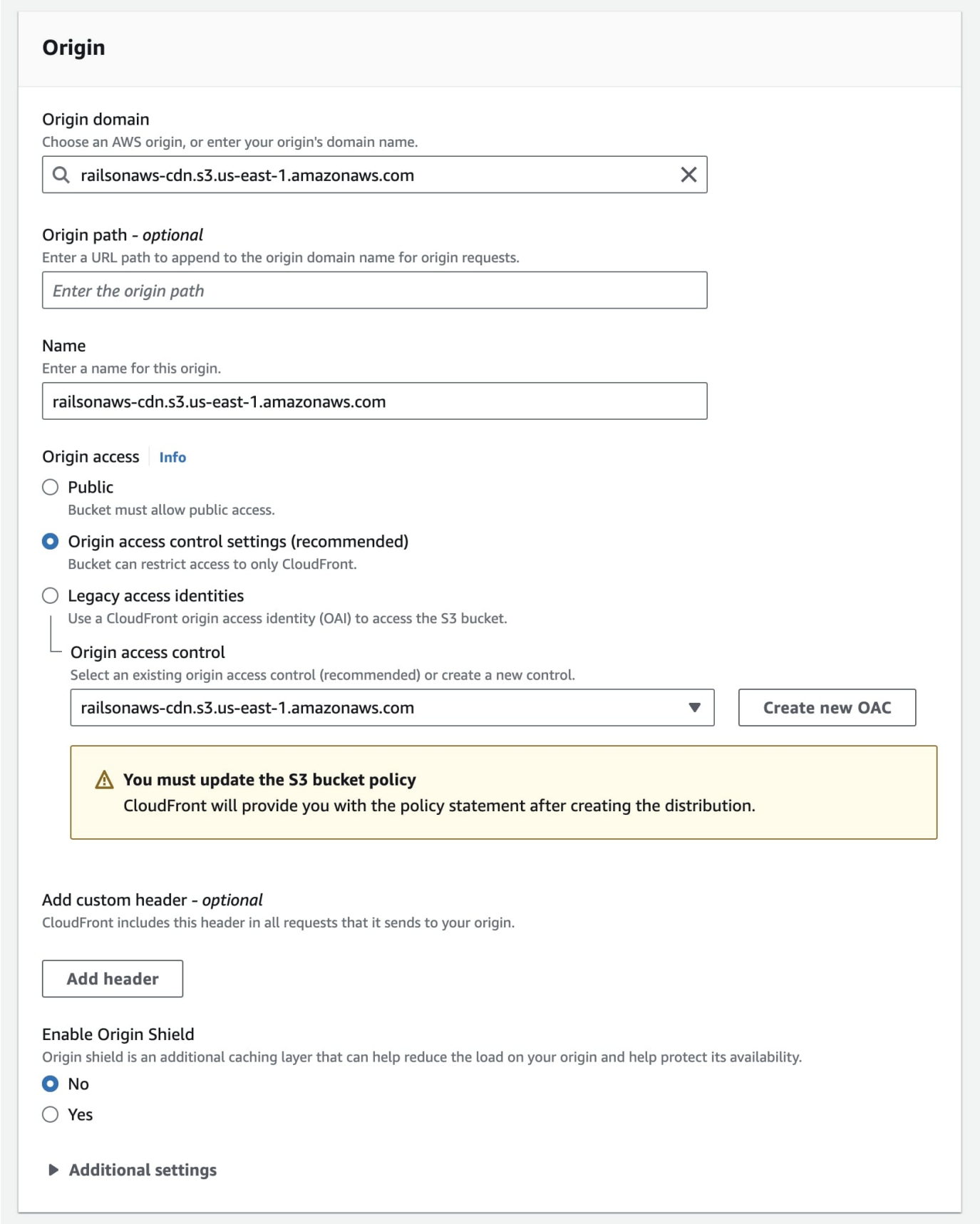
In the S3 chapter, you learned how to provide access to private files on S3 Bucket by

using presigned URLs. Such a solution is efficient when generating temporary access to files only for some people. You can use the signed URLs feature to provide access to private files and benefit from CDN.

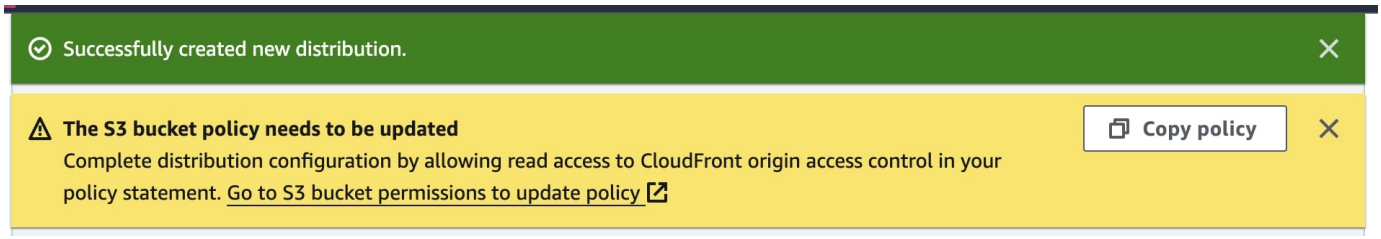
Signed URLs are similar to presigned URLs. They can be valid for a certain period, even for years. However, they are generated differently. Let's start with creating a new private bucket and putting one photo there. I created a bucket "railsonaws-cdn" and uploaded a file named "image.jpg".

CloudFront configuration

Enter the CloudFront dashboard and click on the "Create distribution" button. As the origin domain, select the entry for the S3 bucket that you created in the previous step. In the origin access section, select "Origin access control settings" and click on the "Create new OAC" button; in the opened modal, you don't have to change anything; just submit the form.



The last step is to select “Do not enable security protections” in the “Web Application Firewall” section, and you can create the distribution. After form submission, you will see the information box about s3 permissions:



First, click on “Copy policy” and then on the “Go to S3 bucket…” link. Edit the bucket policy and paste the policy body; save changes.

Generating public key

We will need an RSA key pair to sign the URL that will give access to the private content on S3. You can generate one for the test:

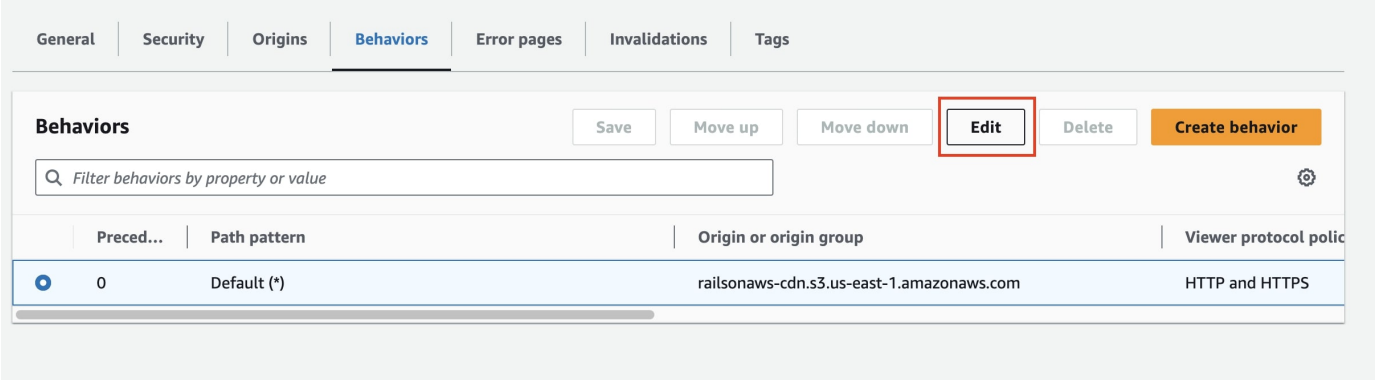
```
openssl genrsa -out test_private_key.pem 2048
openssl rsa -pubout -in test_private_key.pem -out test_public_key.pem
```

Copy the content of the test_public_key.pem file. Open the CloudFront dashboard and click on the “Public keys” link in the “Key management” section on the left side of the screen. You may need to expand the sidebar first. Once you get this, click on the “Create public key” button, paste the public key into the form, give it a name, and submit the form.

From the same sidebar menu, select the “Key groups” link. Click on the “Create key group” button. Select the public key you created, name the group, and submit the form.

Restricting viewer access

Return to your CloudFront distribution page and select the “Behaviors” tab. Select your record and click “Edit”:



Select “Yes” for the “Restrict viewer access”. Select “Trusted key groups” for “Trusted authorization type”. Select your key group from the list. Submit the form.

Generating signed URLs with Rails

Once the configuration is done, we can take care of the coding part and use API to generate a signed URL to access the image in the private bucket. Start with installing SDK:

```
bundle add aws-sdk-cloudfront
```

You will now need to collect the following values: distribution URL, private key content, and the ID of the public key. Go back to the “Public keys” section, and on the list, there should be one public key that you have added. Copy the value from the “ID” column.

Visit the “Distributions” link from the sidebar, which is the list where all your distributions are listed. Copy the value from “Domain name” - it should end with the cloudfront.net domain. Once you have the values, you can generate the signed URL this way:

```
signer = Aws::CloudFront::UrlSigner.new(
 key_pair_id: 'public_key_id',
 private_key: 'private key content'
)

cloudfront_domain = 'https://my-distribution-id.cloudfront.net'
signer.signed_url("#{cloudfront_domain}/image.jpg", expires: Time.now +
```

```
60)
```

Your link will be valid for one minute, and you will benefit from all the features of CloudFront CDN. Such a solution is perfect when you want to serve assets for course and your students are from different parts of the world.

Customizing cache expiration times

By default, CloudFront will keep the cache for 24 hours and then consider it outdated. If you prefer different expiration times, you can either specify the expiration for a group of files or per single file.

Custom expiration time single files

If you want to expire the cache after 60 seconds for a given file, you must set the `Cache-Control` header to `max-age=60` as metadata on S3. You can do this using API or manually from the s3 dashboard.

Custom expiration for multiple files

To change the expiration time for all files or files that match the given pattern, open the CloudFront dashboard, select your distribution, and click on the “Behaviors” tab. Select the existing behavior and click edit.

Scroll down to “Cache key and origin requests”. You can select the dedicated policy under the “Cache policy” label. Currently, you only have AWS policies at your disposal, but you can create your own by clicking on the “Create cache policy” link.

In the cache policy form, you can set custom headers and specify the expiration time for the files.

AWS Transfer Family

AWS Transfer Family is a service that you will find helpful when you need to upload files or your application's partners need to upload files via SFTP, FTP, FTPS, or AS2 protocols. Traditionally, you can build an application page where files can be uploaded, but this is not the perfect solution in many cases. With Transfer Family, you can provide alternative ways to upload files and authenticate users using SSH keys.

In this chapter, you will learn the minimum required to manage the Transfer Family 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
- **Pricing** - I will discuss the general pricing for the service
- **Configuration** - I will walk you through the configuration process of the service to make it ready for your Rails application
- **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 API for the service to create and update users via the Rails application

Unfortunately, this service is not included in the free tier, but it's cheap enough to play with it.

Explanation

It's a common feature that allows users to upload files into the application using a form. However, sometimes, you have to deal with organizations or businesses that follow different procedures regarding data manipulation. It's not always about the law; sometimes, it's about the automated data flow.

To handle custom scenarios, AWS Transfer Family has your back. You can treat this service like an SFTP, FTP, or FTPS protocol that automatically transfers every file to an S3 bucket. You can also configure users via API, add SSH keys to protect the endpoint, and configure isolated directories in the bucket.

I recently used this service for a company that used to receive reports from partners via FTP. Once the reports were uploaded, the company had to update the data using an external API.

To truly understand the configuration of the service, familiarize yourself with the following terms:

- **Protocols** - the way your users can upload the files
- **Identity provider** - the way your users are authenticated. You can keep the authentication data on AWS or use different sources
- **Endpoint** - the address your users will use to connect to upload the files. It can be an endpoint generated by AWS, your domain or subdomain, or a static IP address

Once the configuration is prepared correctly, you can connect, for example, to ssh using `ssh -i your-key username@server-endpoint` when using the terminal.

Pricing

The pricing structure is simple. For each protocol enabled on your server, you will pay an hourly rate depending on the selected region. For the Frankfurt region, it's $0.30, meaning you will spend around $216 for the whole month.

You will also pay for each GB of data transferred into and out of the server. For Frankfurt, it's $0.04 per GB. There is an exception for the AS2 protocol, where you pay for each message. The AS2 protocol is usually used to transfer XML documents between companies safely.

Configuration

The configuration process is straightforward, as you don't have to select instance size or anything like that. AWS fully manages the service, so you only care about passing the proper credentials and uploading files. All of the rest happen without your action.

Type "Transfer Family" into the search bar and click on the first result. On the right side of the screen, click on the "Create server" button; it will render the configuration form that consists of a few steps.

Protocols

You have four protocols at your disposal. You can select all, but be aware that you will pay separately for each protocol, so if any of the protocols listed is optional, don't select it. For this book, I will go only for the SFTP option.

Identity providers

There are three options for the Identity provider. You probably want to select "Customised identity provider" if you run a company and authenticate your employees. If you go with "Service managed", all accounts will be stored on AWS, and it will be effortless to manage them using standard API.

I'm using the service-managed option for this book to demonstrate how to create, update, and delete users and accounts using Ruby SDK for AWS.

Endpoint

If you don't care about static IP, you can go for "Publicly accessible" and AWS will provide you with an endpoint and give you the ability to define the domain. If you need an IP address that won't change because your partners whitelist it, go for the "VPC hosted" option. Even with the VPC option, it will be easy to configure the endpoint as you just have to select VPC from the list and then assign an Elastic IP address.

I will select the "Publicly accessible" option as it's the fastest way to start playing with the service.

Domain

After selecting the endpoint, you must choose which storage service your files will be uploaded to. You have two options: S3 and EFS. I'm going to use S3, as we discussed it before. EFS is rarely used with Rails applications.

Additional details

On this screen, you don't have to update any configuration. AWS will create a new log group for the service so we can check the connection attempts for each configured user. You can scroll down and click "Next".

On the last confirmation screen, scroll down and click “Create”. Once you have done that, you must wait a few minutes for the server to start.

Permissions

Before we can upload our first file, we must create an S3 bucket to which the files will be uploaded. We must also create a role with proper permissions to attach to each newly created user in the Transfer Family service.

Go ahead and create a new bucket on S3. You should be able to do this after reading the chapter about s3. Note the name of the bucket, as we will need it.

Policy for accessing files in bucket

Type “IAM” into the search bar and click on the first result. On the menu rendered on the left side of the screen, click on the “Policies” option and then the “Create policy” button in the top right corner.

In the top right corner, select the “JSON” option, paste the following policy body, and replace “bucket-name” with the name of your bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowListingOfUserFolder",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::bucket-name"
      ]
    },
    {
      "Sid": "HomeDirObjectAccess",
```

```
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:GetObjectVersion"
      ],
      "Resource": "arn:aws:s3:::bucket-name/*"
    }
  ]
}
```

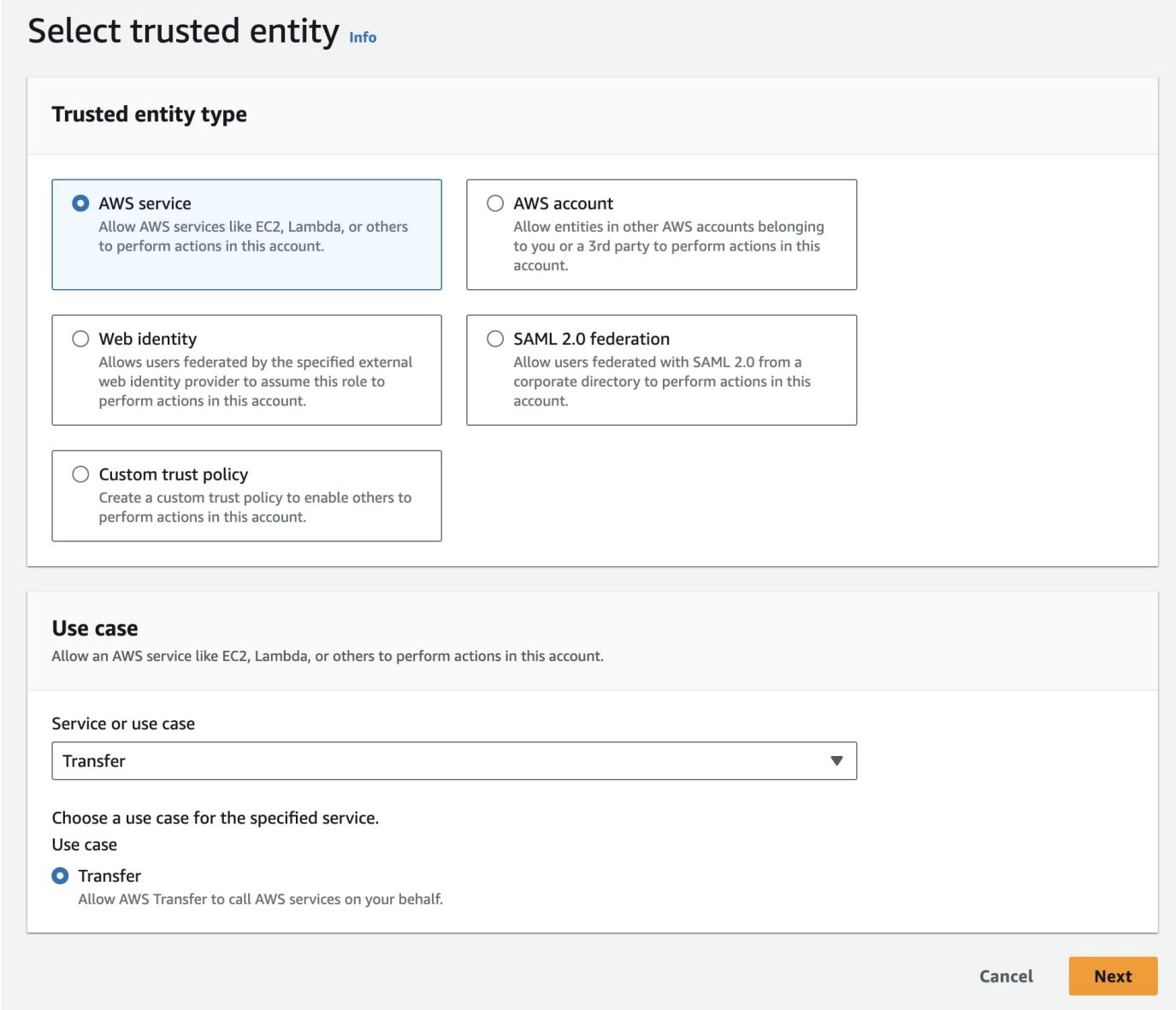
This policy grants permissions to manage files inside the bucket. Click “Next” and add a name for your policy, then click the “Create policy” button.

A role for the Transfer Family server

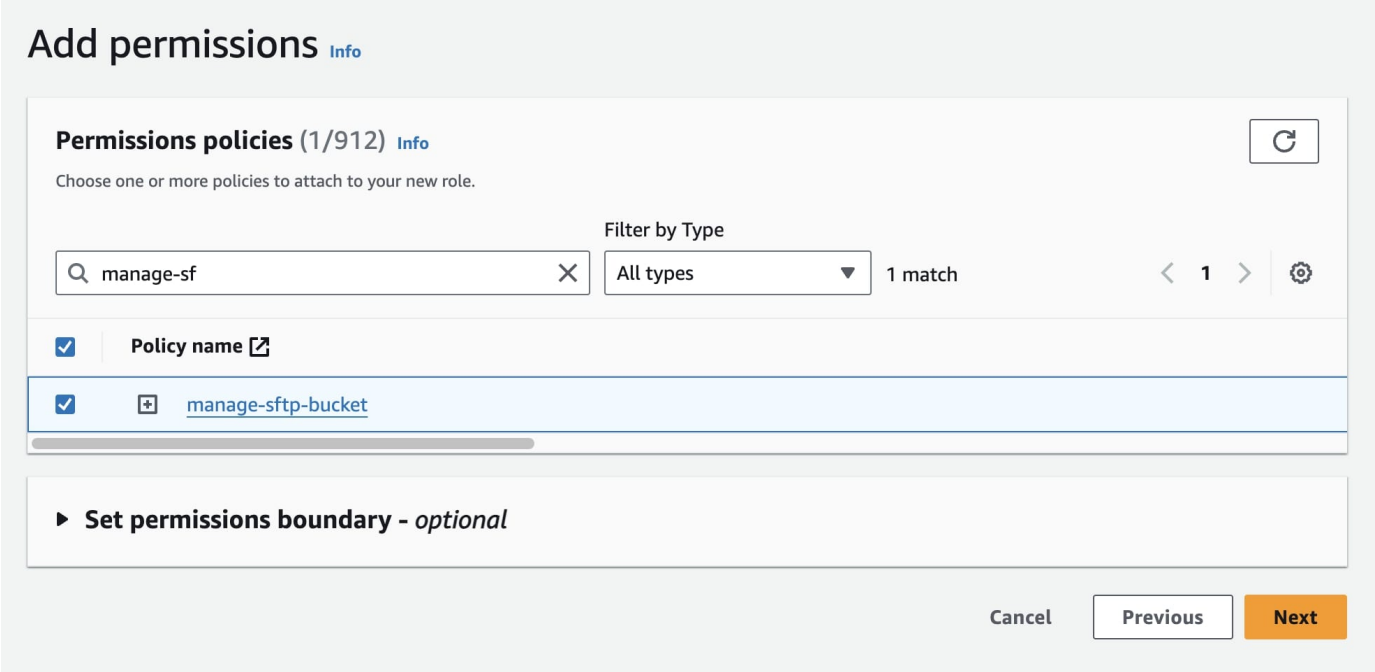
We created the policy, and now it’s time to create a role that we will attach to each user created on our server so it’s possible to upload files to our bucket via this user.

Type “IAM” into the search bar and click on the first result. On the menu rendered on the left side of the screen, click on the “Roles” option and then the “Create role” button in the top right corner.

Select “AWS service” as the Trusted entity type, and Transfer on the list for “Service or use case” and click “Next”.



In the permissions search bar, type the policy name you created in the previous step. Mark the position on the list and click “Next”.



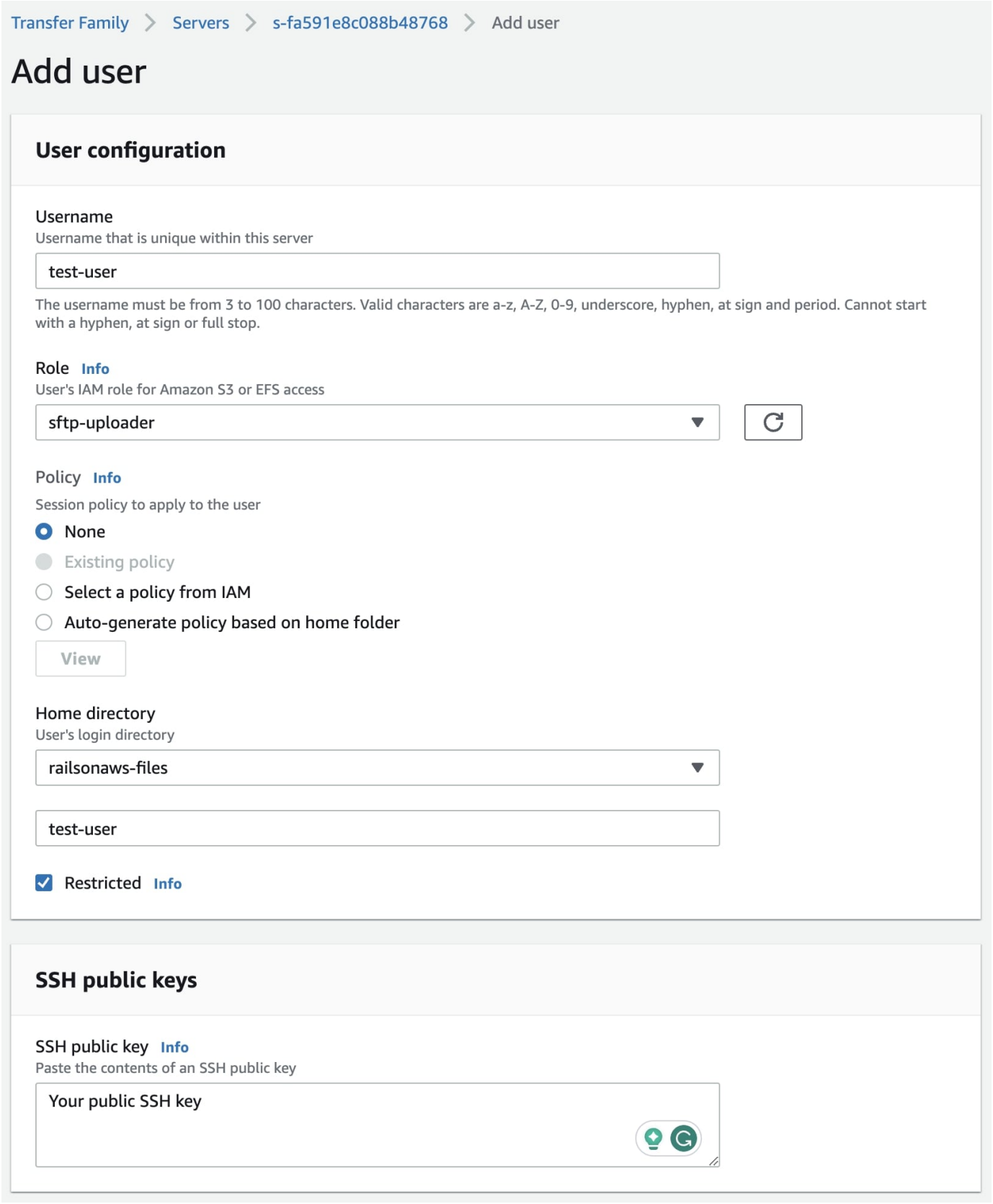
Provide the role name and click “Create role”. We just configured all we needed to add the first user and upload our first file.

Adding the first user and uploading the first file

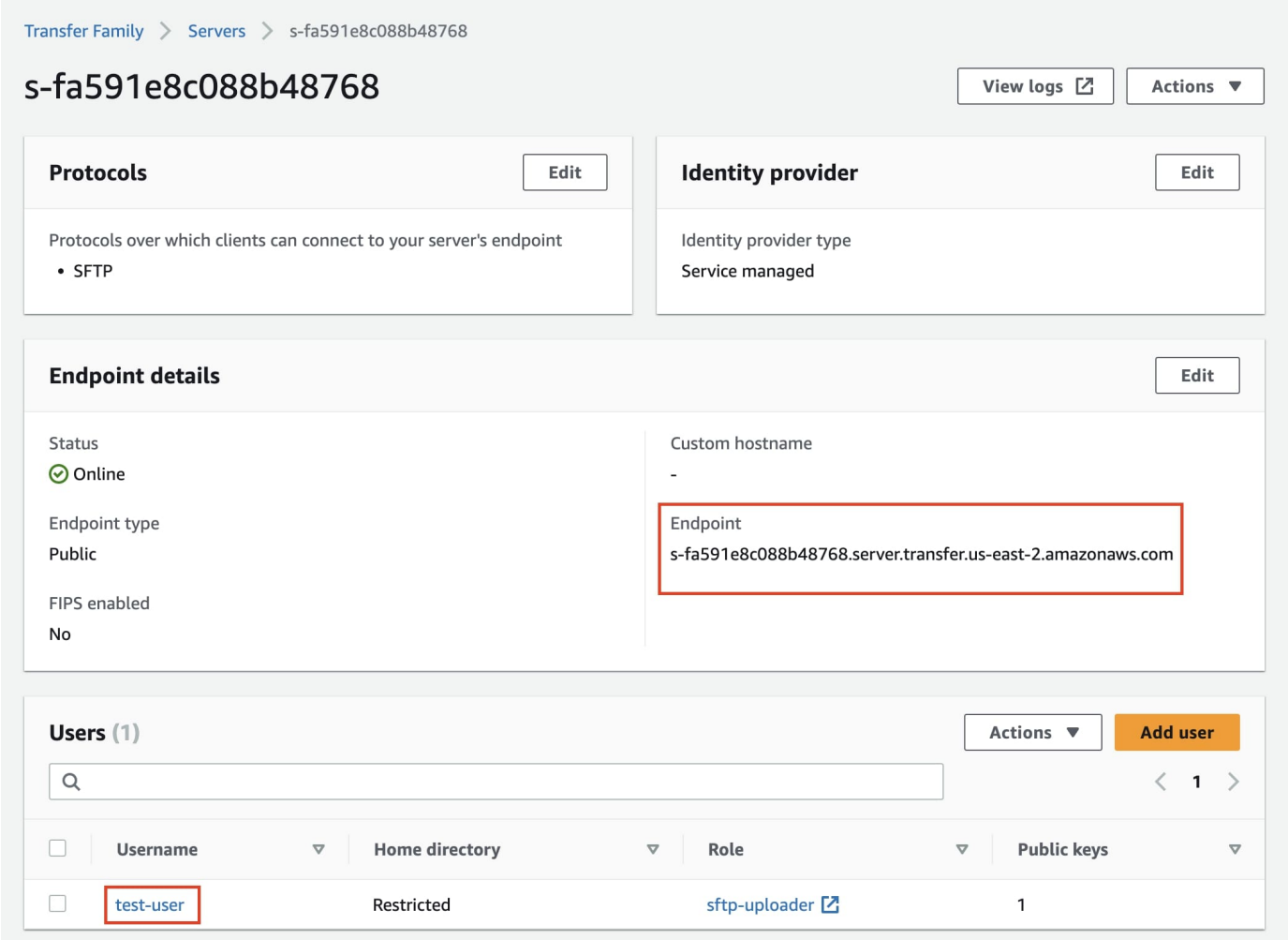
Return to the “Transfer Family” dashboard and click on your server on the list. Scroll down to the users box and click the “Add user” button. You have to provide the following information for the new user:

- **Username** - this is the identification for your user
- **Role** - the role you created in the previous step
- **Home directory** - select the bucket you created before
- **Optional folder** - below the home directory select list, you have an input to type the folder name for your user. I put here the name of the user as I want each user to be a separate directory in the bucket
- **Restricted** - if you check this option, your user won’t be able to see files in other directories
- **SSH public key** - paste your SSH public key so you can test the service and upload the file

Once you provide all the details, click the “Add” button.



You can now copy the server address and your username to connect using the terminal:



If you are on a Mac (on Linux as well), you can use the sftp command line program to connect to the server. The command will look like this:

```
sftp -i path_to_private_ssh_key username@aws-endpoint
```

You should be able to connect. Let's create a text file and then upload it to our sftp:

```
echo "Hello world" >> test.txt
sftp -i path_to_private_ssh_key username@aws-endpoint
sftp> put test.txt
```

You can now navigate to the bucket on S3, and you should see a new directory for your user and the test.txt file inside. Congratulations, you just created your first Transfer Family user and uploaded the first file! It's time to do the same but use Ruby SDK in the

Rails application.

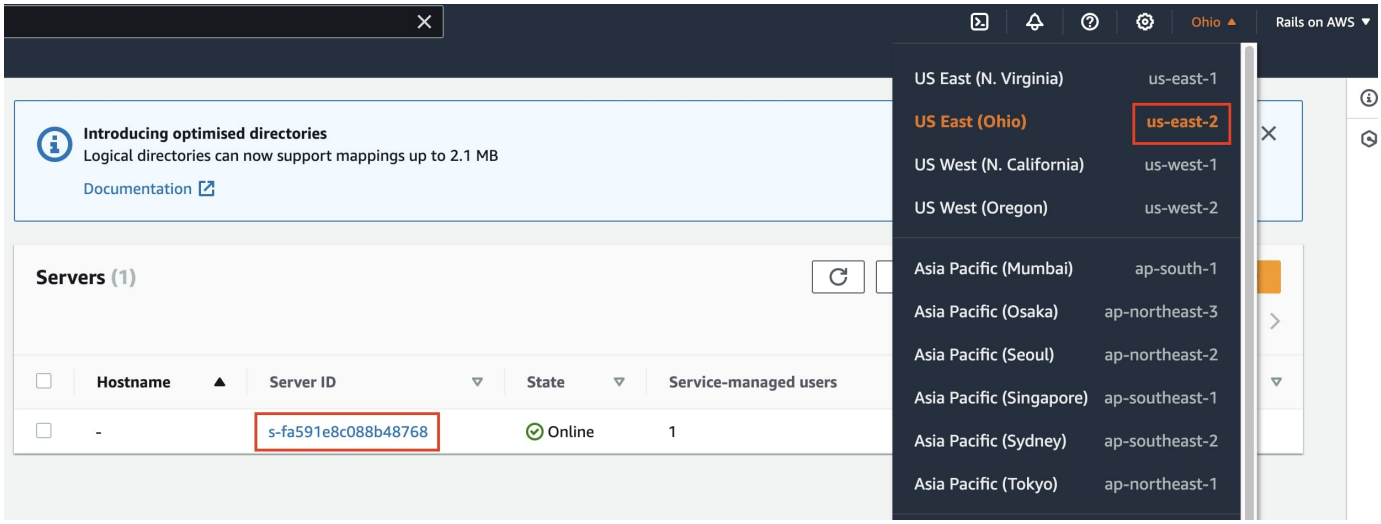
Development

It's finally time to write some code, but right before it... we need to create proper API keys with permissions. As always!

Creating credentials and permissions for Ruby SDK

Suppose you read the chapter about S3, you already know how to create a new user and generate API keys. If not, take a step back and familiarize yourself with this chapter. You can either create a new user or use the existing one.

We will need a policy allowing our user to manage the Transfer Family users via API. First, go back to the Transfer Family dashboard, copy the server ID, open the region select list, and copy the region code and the account ID:



In my case, I can see the account ID by clicking on the "Rails on AWS" account name in the top right corner. Note the following ARN address:

```
arn:aws:transfer:{region}:{account-id}:server/{server-id}
```

Now, open the IAM dashboard and the roles list. Select the role you created for Transfer Family and copy the value below the "ARN" label. It should have the following format:

```
arn:aws:iam::{account-id}:role/{role-name}
```

Having these two ARNs' values, we can create the proper policy for our API access. Open the IAM dashboard and select “Policies” from the list on the left side. Click on the “Create policy” button and select the JSON option. Paste the following policy body and replace {transfer-family-arn} and {role-arn} with your values:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
          "transfer:*"
      ],
      "Resource": "*"
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "transfer:*",
      "Resource": "{transfer-family-arn}"
    },
    {
      "Effect": "Allow",
      "Action": [
          "iam:GetRole",
          "iam:PassRole"
      ],
      "Resource": "{role-arn}"
    }
  ]
}
```

Click “Next”, give your policy name, and click “Create”. From the list on the left side, select the “Users” link and find your user on the list. On the top right side, click on “Add permissions” and then select “Attach policies directly”, find your policy on the list, mark it as checked, click on “Next” and then “Add permissions”.

Installing SDK library

We have the proper permissions set and API keys generated. We can now install the SDK for the Transfer Family service:

```
gem install aws-sdk-transfer
```

You can now initialize the client the following way:

```
require 'aws-sdk-transfer'

client = Aws::Transfer::Client.new(
  access_key_id: 'access-key',
  secret_access_key: 'secret-key',
  region: 'transfer-family-server-region'
)
```

Creating a new user

Let’s create the test-user again, but now with the API:

```
user_name = 'test-user'
files_bucket_name = 'bucket_name'

client.create_user(
  role: 'aws-transfer-role-arn',
  home_directory_type: 'LOGICAL',
  server_id: 'aws-transfer-server-id',
  user_name: user_name,
```

```
  home_directory_mappings: [
    {
      entry: '/',
      target: '/#{files_bucket_name}/#{user_name}'
    }
  ]
)

response = client.import_ssh_public_key(
  server_id: 'aws-transfer-server-id',
  ssh_public_key_body: 'ssh_public_key',
  user_name: user_name
)

response.ssh_public_key_id
```

You can now upload files to the server using the test-user user and proper SSH key. If you want to delete the SSH key assigned to the test-user, save somewhere value from response.ssh_public_key_id.

Updating existing user

If you want to update the SSH key assigned to the existing user, you have to know the public key ID of this key. With this information, you can execute the following code:

```
client.delete_ssh_public_key(
  server_id: 'aws-transfer-server-id',
  ssh_public_key_id: 'current-key-id',
  user_name: user_name
)

client.import_ssh_public_key(
  server_id: 'aws-transfer-server-id',
  ssh_public_key_body: 'new public ssh key',
  user_name: user_name
```

```
)
```

Deleting user

This is the most effortless action to perform. You just have to provide the server ID and the user name:

```
client.delete_user(
  server_id: 'aws-transfer-server-id',
  user_name: user_name
)
```

Important information - even if you delete the user, his directory and files will still be present on s3, so if you don't want to keep them, you have to remove them explicitly. The same applies to deleting the whole server; all previously uploaded files will still be on s3.

P.S.

Delete the Transfer Family if you have finished testing.