AWS Serverless Patterns – S3

Handling documents in a private S3 bucket

Ivan Castanon

Jim Ladd

July 22, 2021

Contents

Introduction 3

Background 3

Viewing Documents in a Private S3 Bucket 5

Uploading Documents to a Private S3 Bucket 6

Manipulating Documents in a Private S3 Bucket 8

Summary 9

Who We Are 9

References 10

Introduction

While the solution space for software development is constantly updated with new technologies, the requirements and artifacts of the problem space can appear to be timeless. Even with cutting edge technologies, software developers are still burden with vintage entities like document and image files. This paper facilitates the adoption of the AWS serverless stack by describing three real-world patterns involving the document lifecycle and the AWS Lambda and S3 services. The patterns cover uploading documents to a private S3 bucket by multiple users, allowing read access to a subset of the user population, and manipulating the documents in a serverless realm.

Background

SOFWERX is currently migrating its public facing website from WordPress to the AWS serverless stack. One of the objectives of this project is to forge the AWS serverless technologies into a simple yet robust platform [1]. The major services of this architecture include:

- ***React*** – A common JavaScript framework.
- ***Amplify*** – This is a set of tools to ease the development of mobile and web applications. This project uses Amplify to send the HTTPS requests to the API Gateway.
- ***Cognito*** – This service facilitates the process of user sign-up, sign-in, and access control.
- ***S3*** – The Simple Storage Service is AWS's object storage service.
- ***CloudFront*** – This service provides a fast content delivery network.
- ***API Gateway*** – The gateway service provides a "entry point" to the other AWS services.
- ***Lambda*** – This service allows code to be executed on demand and without managing servers.
- ***DynamoDB*** – This is a key-value and document high performance database.

A diagram of this architecture is shown below:

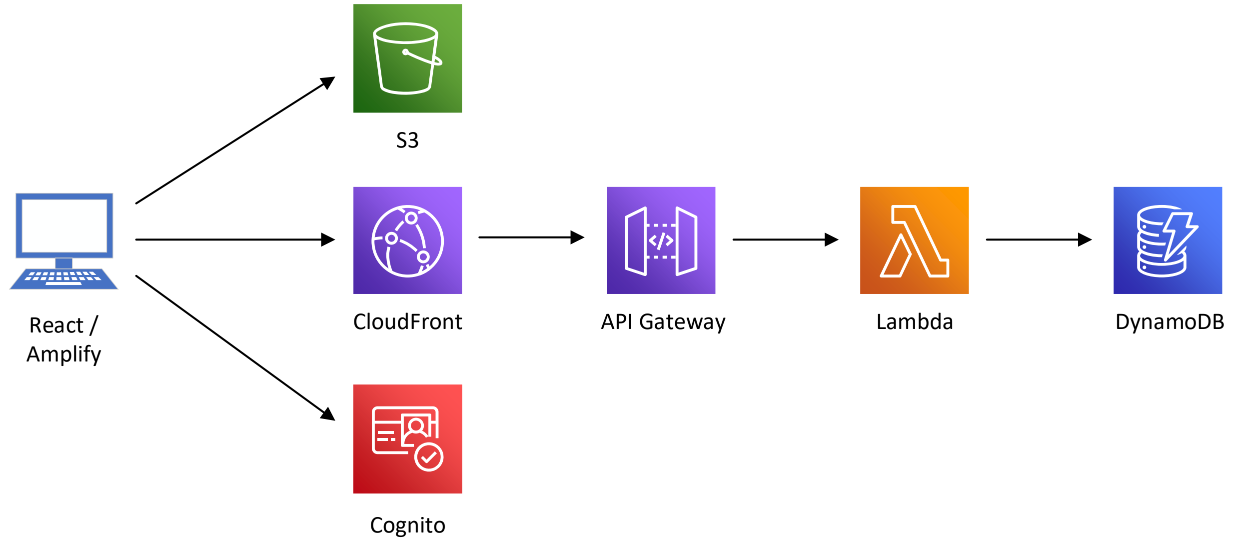

Figure 1 - SOFWERX web application architecture

One of the requirements imposed on the web site is to restrict access to a set of documents. While our online information is not classified, we want to limit the access to our client and other government agencies. The primary motivation is to ensure a vendor's intellectual property is not available to competitors.

Another requirement is to alleviate the intended users from individual login credentials. Our solution is to incorporate the DoD Common Access Card (CAC) in order to validate the users and facilitate their access to the protected documents [2]. The CAC is a "smart" card and provides standard identification for active and reserve military personnel, Department of Defense civilian employees, and eligible contractors. This card is close in size to a credit card and enables physical access to controlled facilities. It also provides access to DoD-related computer network and systems.

The web security validates the CAC of the user and, if it is valid, logins into the web site as a generic CAC user. We do not track individual user information since a valid CAC is sufficient authentication for SOFWERX's needs. If a user does not have a CAC, he/she may request a dedicated user profile to be created and can login to the web site with the granted credentials. A diagram of the architecture is shown below:

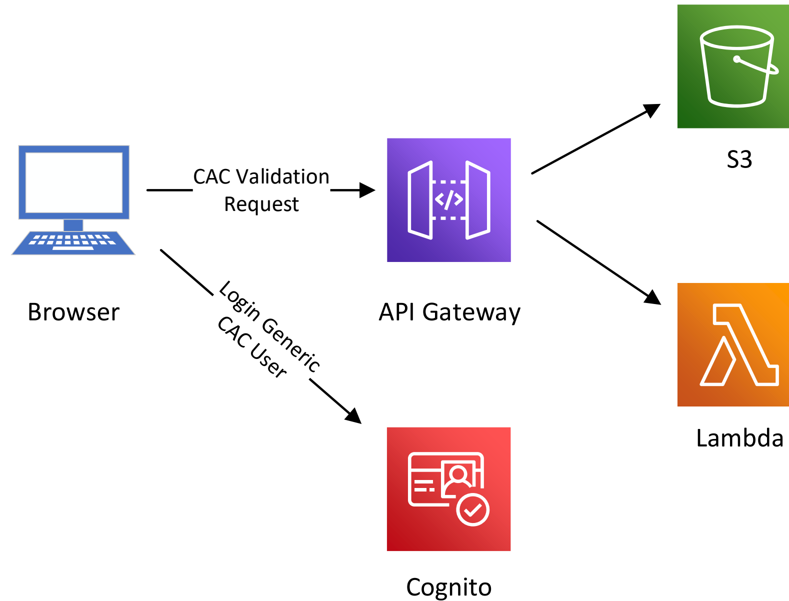

Figure 2 - SOFWERX CAC security architecture

In summary, we have documents, in pdf and jpeg formats, that can be viewed by a subset of our users and are not accessible by the general public. We also want a SOFWERX employee to be able to upload new documents via our "admin" web pages. Lastly, after a new pdf file is uploaded to S3, the text needs to be extracted from the document and saved in DynamoDB for further processing and analysis. During the early development of the new website, we identified three patterns that guided our design to meet these requirements. The remainder of this document describe these patterns.

Viewing Documents in a Private S3 Bucket

The first pattern describes how to view restricted files in S3. This approach involves the creation of a private S3 bucket to store the documents. The bucket name and file name are stored in DynamoDB. The user is validated by the web client using Amplify and Cognito. The web client issues a request to the API Gateway where it is authorized and, if approved, passed to the corresponding Lambda method. This method retrieves the S3 bucket name and file name from a table in DynamoDB. With this data, the Lambda method generates a *presigned URL*. A presigned URL is generated by an AWS user who has access to the object (in our case, the Lambda method) and will use the credentials of that user [3]. A presigned URL remains valid for a limited period time which is specified when it is created. The presigned URL is returned in the response to the web client and can be used to access the S3 object. Our documents are jpeg and pdf formats but this approach should work for most document types. The following diagram shows the sequence of invocations for using a presigned URL.

SOFWERX

Browser
Request
presigned URL value
API Gateway
Invoke
presigned URL value
Lambda
Get S3 Bucket & Object Names
DynamoDB
Request presigned URL
S3

Figure 3 - AWS presigned URLs

The value of a presigned URL of a pdf document is shown below:


```
https://swx-project.s3.amazonaws.com/1626182849068-Encrypted-
Keyboard.pdf?AWSAccessKeyId=ASIASBWZIDZ7E2U7WK7Q&Signature=k6SOoJAIytYypA
JUD0QROt5y8Kg%3D&x-amz-security-
token=IQoJb3JpZ2luX2VjEN7%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLWVhc3QtMSJ
HMEUCIQD5SGe2SE%2BOxTHX0Pu2mKV3bLv0bi4gKRdOQ%2B3QViv%2BIwIgLbn74Sud1Y99u%
2FHY%2Fi4VN6kCjiPwafE%2FEDcudoL3l9oqtAII1%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2
FARACGgwxNDExMTU3OTMwMjIiDNqCFSLYe%2BAmj%2BPw8iqIAl1TJCV2fSmdZ8cmkNa2%2BY
zO%2FlS1OWQHbk%2BzNidCiWE0N9kxNstkHkndUOev9HORp1gVqFqeA%2F6y84LCp%2FFQ6Le
%2BVsI1k2ZVpMEOKI3V6wCuxtdPhCgGsX07Uo70BrYAjNYd%2F1vrWfBTXhv778qfdYslF78x
qkE188kdJRAtDyaBABpaRgmg9MYiE4vrQA66Lsl8W6A%2FFqYCjS5AzcxpPcXcbD9NrHZmXxN
77x0avCu2agTThQe21vZpoWnBBuyniVJC5HLDtxC4Waa0TZ8qXBQcOgmiGNlN59%2BRiLcFHt
cNqEZB%2B9BAaIp0ms7LuEPfB4kd5KCoWYhx0ZytuByCdQ%2FJ5DUx9hEo1TDS6uWHBjqaAWT
aYaGyyGJaWu2xKLEnXbSjXRC9GYwDkvs7eb%2FimO2aGZftyDvAULgFEOflx05MxDFlaoNeb6
dDulQUWb6TiDrYD3BeN58TTwKpt8cGuid0EAfuGqVzdsBCI%2Br7T9hY2%2BOvbRBTLejA1ND
A%2Fuiov0uEa%2BNqgTfd2T23vDJYnzp3g%2BGoXf1cWOG%2F7UFEQgzVyk7IsJeOmFVZCis%
3D&Expires=1626964835
```


Uploading Documents to a Private S3 Bucket

This S3 pattern is concerned with uploading documents by different users to a private bucket. When a web client uses the Storage feature in the *aws-amplify* library to upload a file to a private bucket, AWS creates a folder in the bucket with the name of "private". In this folder, a sub-folder is created with the name of the "Identity ID" of the user who is uploading the file. For example, the following diagram shows the file structure of a private bucket with uploaded documents (i.e., my-file-100.pdf, my-file-200.pdf, and my-file-AAA.pdf) from two different users.

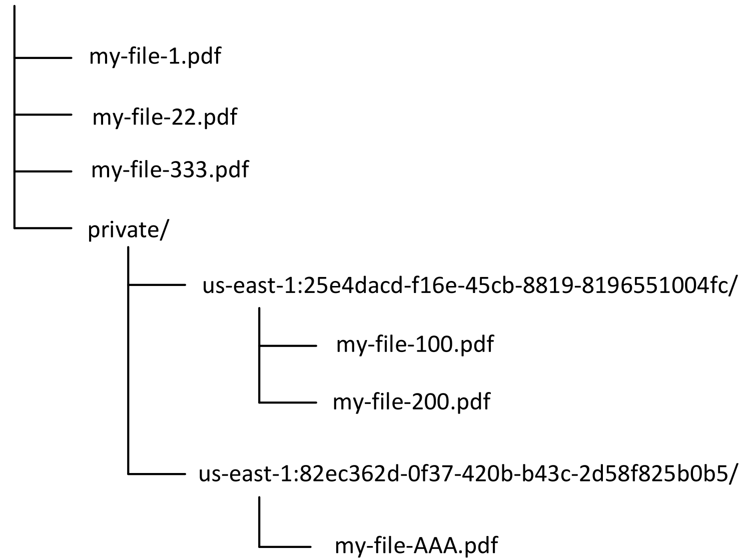

Figure 4 - S3 bucket with uploaded files

The key concept of this pattern is that after being uploaded to S3, the files are moved from their initial path (i.e., bucket/private/<Identity ID>) to a different location. In the example above, the three newly updated files would be first copied to the top directory and then deleted from their original folder. In our architecture, the event sequence between the AWS services is shown below:

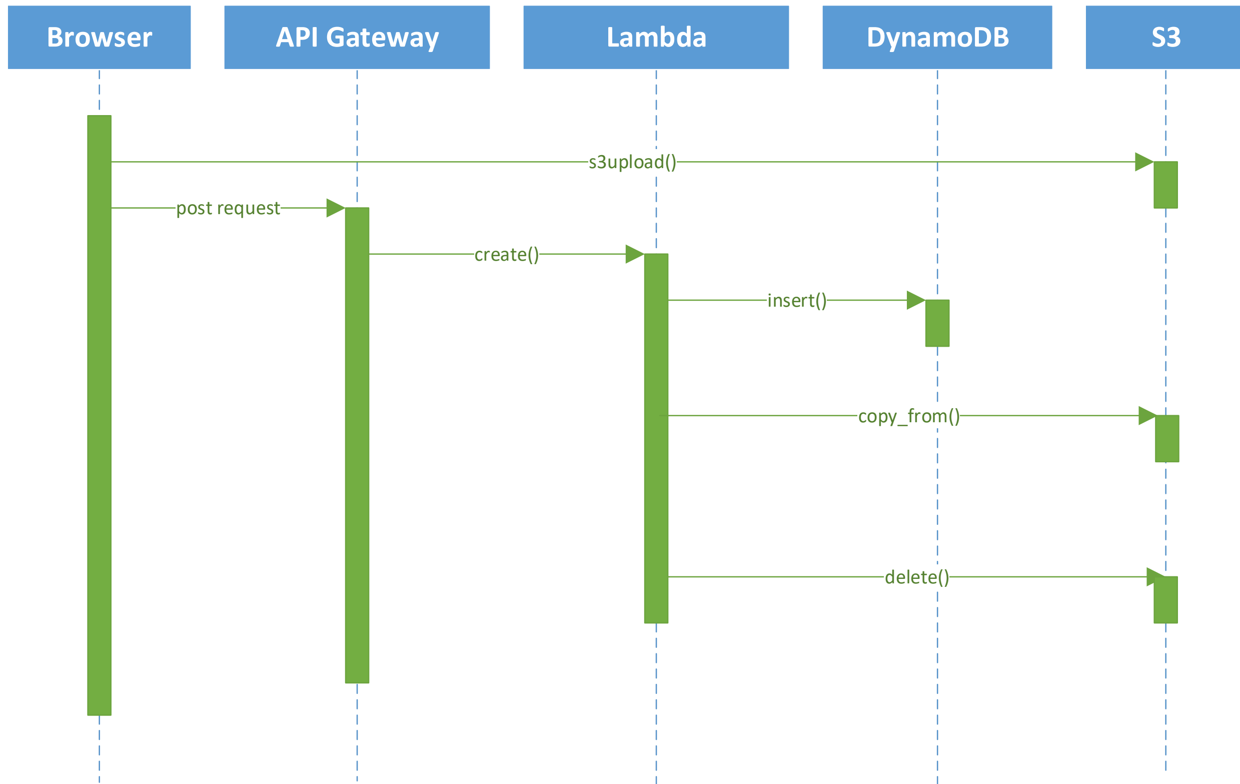

Figure 5 - Event sequence for uploading documents

Manipulating Documents in a Private S3 Bucket

This pattern deals with reading and writing documents residing in a private bucket. In our website, the text of the uploaded pdf file needs to be stored in a DynamoDB table for analysis and additional processing. We decided to extract the textual content after the file is uploaded to S3 by the administrator.

While the initial inclination is to use traditional file-based operations to extract the data, we believe this violates the serverless paradigm. A file is an ordered and named collection of bytes that has persistent storage. When you work with files, you work with directory paths, disk storage, and file and directory names. In contrast, a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums (for example, disks or memory) [4].

A 'stream' is internally nothing but a series of characters. Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation [5].

We strive to construct our serverless systems to be independent of operating system or container characteristics. Since streams are less coupled to the underlying operating system, we use them when manipulating objects stored in S3 buckets. For example, the Python code used to extract the text from a

pdf file on an S3 bucket is shown below. This code is embedded in a Lambda function that is invoked after the file has been uploaded.

```
import boto3
import pdfplumber
from io import BytesIO


def get_pdf_text(bucket_name,file_name):
    text = ""

    s3 = boto3.resource('s3')
    obj = s3.Object(bucket_name, file_name)
    fs = obj.get()['Body'].read()

    with pdfplumber.open(BytesIO(fs)) as pdf:
        for page in pdf.pages:
            text = text + '\n' + page.extract_text()

    return text
```

Figure 6 - Python code snippet with streams

Summary

As the world moves ever closer to a paperless environment, it remains far from being document-free. Documents have been, are, and will be important instruments of information. Even the latest software technologies like the AWS serverless stack must accommodate documents. This paper presents three real-world patterns that make document handling in AWS a little less painful.

Who We Are

SOFWERX is a non-profit entity that accelerates evolution of the Warfighter through technology discovery, engagement, development, and transition. SOFWERX was created under a Partnership Intermediary Agreement, established in September of 2015, between DEFENSEWERX and the United States Special Operations Command (USSOCOM).

Ivan Castanon is a candidate for Bachelor of Science in Computer Engineering at the University of South Florida. At USF, Ivan worked in the Neuro-Machine Interaction Lab as a Research Assistant. He contributed to this project during the Summer 2021 internship at SOFWERX. His LinkedIn profile link is https://www.linkedin.com/in/ivan-castanon-740155200/

Jim Ladd is a Senior Software Architect and IT Manager at SOFWERX where he leads the software engineering, web development, and system administration teams. Jim has been developing software solutions for over 35 years. Before joining SOFWERX in 2019, Jim was CEO and Principal Consultant at Wazee Group, a niche consulting company, for 20 years. His LinkedIn profile link is https://www.linkedin.com/in/jim-ladd/

References

[1] S. S. &. J. Ladd, "Fast Path to AWS Serverless Applications," [Online]. Available: https://sofwerx.github.io/docs/ServerlessStackApp.pdf.

[2] J. Ladd, "CAC in the Cloud," [Online]. Available: https://sofwerx.github.io/docs/CACInTheCloud.pdf.

[3] "Presigned URLs," [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-presigned-urls.html.

[4] Microsoft, "File and Stream I/O," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/io/.

[5] M. Mukherjee, "A Gentle Introduction to C++ IO Streams," [Online]. Available: https://www.cprogramming.com/tutorial/c++-iostreams.html.