Managing an Amazon S3 bucket via SFTP using AWS Transfer Family

14 May 2021 - 7 min. read

Amazon S3 | AWS Lambda | AWS Transfer Family | SFTP

Protocols for file transfer to remote servers have been around since the dawn of computer networking. FTP (exactly, File Transfer Protocol) is one of the fundamental building blocks of the internet. Developed by an MIT student in the early 1970s, FTP has become the standard for remote file transfer and management for decades.

Over the years, FTP has been upgraded to provide several advantages. Most importantly, SFTP and FTPS have been developed to supplant the historical protocol by establishing secure data streams.

The managed AWS service, AWS Transfer Family, provides a fully managed set of resources to support an additional way to transfer files in and out of AWS. This service allows the exposure of a convenient interface to manage objects on Amazon S3 and Amazon EFS using well-known file transfer protocols like FTP, SFTP, and FTPS.

How does it work?

This AWS service allows you to avoid the maintenance hurdles of self-managed FTP servers. In fact, AWS Transfer Family takes care of scaling the underlying EC2 servers granting the right capabilities, keeping the whole service highly available.

For user authentication, AWS Transfer Family allows you to choose between service-managed and custom solutions. The first option, however, while allowing a very quick configuration of the service using Aws generated SSH RSA keys for SFTP Authentication, does not support the integration with existing authentication mechanisms or even plain old username password authentication.. The second option

instead, gives you “carte blanche” when you need to integrate an existing identity provider. For example it is possible to use LDAP or Microsoft Active Directory as IdP, or set up custom Auth systems backed by ad-hoc lambda functions.

As said before, AWS Transfer Family allows access to remote files stored on S3 or EFS by employing FTP, SFTP, and FTPS protocols. It’s important to note that the usage of FTP is not supported for internet-facing workloads, in fact, simple FTP connections are considered insecure due to the plain text transfer of credentials, allowing only VPC mode.

Why do I need AWS Transfer Family?

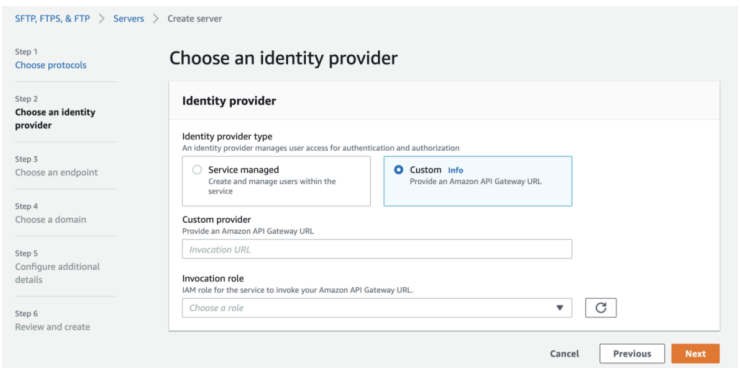
The provisioning of the infrastructure needed to maintain an architecture enabling FTP, SFTP, or FTPS file transfer can be very burdensome in both economic and maintenance terms, AWS Transfer Family allows you to configure new environments or migrate existing ones limiting these concerns.

It must be pointed out that the migration of similar workloads to AWS Transfer Family doesn’t affect the user experience, allowing them to use the FTP clients that they are accustomed to, such as Cyberduck, FileZilla, WinSCP, etc.

Meanwhile, the usage of this service allows you to bring your data into a durable and highly available storage service. As we’ll see in the following part, the adoption of S3, for example, can give space to a significant number of interesting business cases.

Self-managed user authentication

To configure self-managed user authentication in AWS Transfer Family, we’ll need to specify an API Gateway endpoint and a role to invoke it.



API Gateway (configuration steps can be found here) must expose an API backed by an AWS Lambda. This API will be called by AWS Transfer Family to check the credentials of the user that made an FTP request to the service.

The Lambda function will contain the actual logic needed to authenticate the user. Here is where the magic happens, in fact, you'll be able to implement any sort of authentication: from calling external IdPs API, to implementing everything from scratch by employing other AWS services.

To make the developer's life easier, AWS Transfer Family will include the following properties in the payload of the HTTP GET request:

```
{
   "username": "The user's username",
   "password": "The user's password, if empty SSH authentication is u
sed",
   "serverId": "ID of the AWS Transfer Family server",
   "protocol": "FTP | SFTP | FTPS",
   "sourceIp": "IP Address of the client"
}
```

The lambda function, then, should implement the logic to check the credentials received from the FTP service against the selected user store. In case of a successful validation, a particular JSON object should be returned.

More specifically, AWS Transfer Family expects a response with the following schema:

```
{
   "Role": "[REQUIRED] ARN of a role allowing the base access to S3 b
uckets and KMS keys (if needed).",
   "HomeDirectory": "The S3 bucket prefix that the user will be dropp
ed into when they log in.",
   "Policy": "A scope-down policy useful to restrict access to certai
n buckets according to the user's details.",
   "PublicKeys": "If no password was provided (SSH key-based authenti
cation), then the public SSH key associated with the user is returne
d. This is a comma-separated list and can contain two public keys.",
   "HomeDirectoryDetails": "This is useful to define a logical bucket
structure (symlink). In this way it's possible to show in the FTP cli
ent all the buckets to which a user has access, or create logical lin
```

```
ks to folders."
}
```

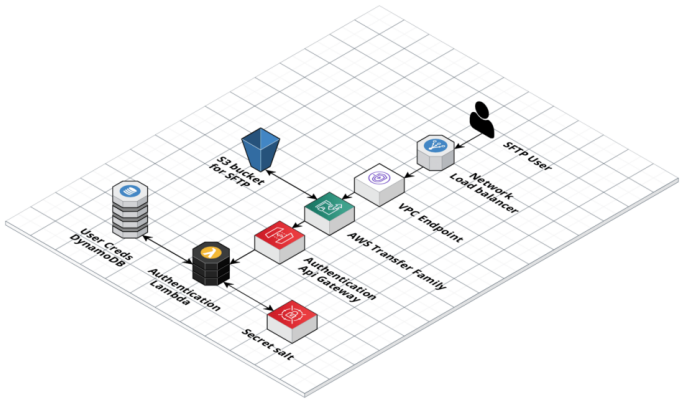
A real-world use case

As mentioned above, AWS Transfer Family allows you to develop and configure custom authentication mechanisms. For example, it would be possible to employ this feature, as we'll see in this chapter, to create a totally new mechanism by using the means that AWS provides.

The objective of this article is to present a real-world example of the migration of an SFTP workload hosted on EC2 to AWS Transfer Family, employing a custom authentication using DynamoDB as a single-digit millisecond latency user store.

The architecture

To achieve the aforementioned objective, we first need to delineate the needed infrastructure building blocks.



As you can see from the schema, we had to introduce a couple of unexpected services such as a Network Load Balancer,and a VPC endpoint. These two components are needed to make sure that the exposed IPs are fixed and will never change. In fact, in this way it's possible to allocate an Elastic IP, also enabling you to use an IP address that you own (BYOIP). This feature is commonly required for specific problems with firewall rules.

Apart from the new components just cited, when an FTP (or SFTP/FPTS) request is sent by the user's client, it is routed through the VPC to the AWS Transfer Family service.

At this point, the scenario described in the Self-managed user authentication paragraph takes place.

In fact, AWS Transfer Family will forward the information required for the user's authentication to the internal API Gateway, that will invoke the authentication lambda.

What happens in the authentication Lambda

As said before, the authentication process will use the DynamoDB table as a user store. In this table, in fact, some important fields will be stored. Its records, will reflect the following structure:

```
{
   "hashPassword": "Used as partition key, is an hash of a combinatio
n of the salt, password and username",
   "username": "The username by which a user can authenticate with a
 password",
   "bucket": "Name of the bucket to which the user can connect",
   "home": "Home directory of the selected bucket",
   "iamPolicyArn": "ARN of the scope-down policy of the user"
}
```

First of all, the hash of the combination of the received credentials plus a fixed secret salt (stored in AWS Secrets Manager) is computed.

The choice of the *hashPassword* field as the partition key allows the direct access to the information of a user. Most importantly, a query will return the specific record of a user if the credentials are correct, otherwise an empty response is received. This, in fact, results in a very fast query against the DynamoDB table, granting a pretty-close-to-instant authentication.

When a successful authentication occurs, the lambda will generate an AWS Transfer Family compatible object and return it to the service.

```
if user:
   print("Authenticated")
   home_directory = "/" + user['bucket'] + "/" + user['home']

   policy_arn = user['iamPolicyArn']
   policy_info = iam.get_policy(PolicyArn=policy_arn)
```

```
    policy_document = json.dumps(iam.get_policy_version(
        PolicyArn=policy_arn,
        VersionId=policy_info['Policy']['DefaultVersionId']
    )['PolicyVersion']['Document'])

    role_arn = "arn:aws:iam::" + \
        os.environ['AWS_ACCOUNT_ID'] + ":role/" + \
        os.environ['USER_S3_ACCESS_ROLE']

    body = {
        'Role': role_arn,
        'Policy': policy_document,
        'HomeDirectory': home_directory
    }

    print("Body: ", body)
    return body
 else:
    # Answer is empty if not succeeded
    print("Not Authenticated")
    return {}
```

Takeaways

Although many protocols for file transfer exist, the historical FTP protocol is still widely used. The creation or migration of these workloads is significantly simplified by the adoption of managed services like the AWS Transfer Family.

In addition, it's important to note that the integration of serverless managed services, both for storage (S3), and for the user management and authentication logic (e.g. ApiGateway/lambda and DynamoDB) allows the service to scale to adapt to any workload and user base size.

What do you think about AWS Transfer Family resources? Did you already put them into action? Tell us about your projects in the comments and see you in 14 days on **#Proud2beCloud** for a new article!

Christian Calabrese

DevOps Engineer and Cloud-native Applications Developer @ beSharp. HiFi enthusiast and hardened videogames player!
