Craft Data Synchronization Solution

Name: Alex Stefan

Solution for Craft Data Synchronization Solution with AWS

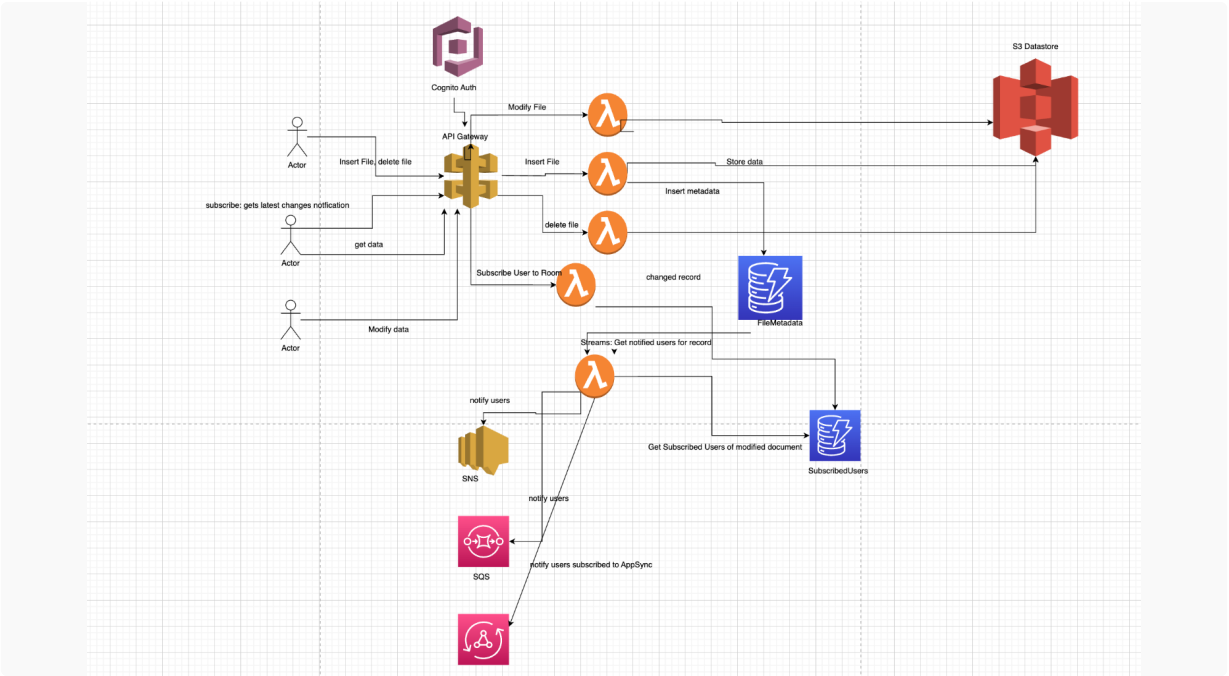
Repository: https://github.com/alexandrustf/realtime

Overview

I have built a system which is able to:

- Insert File by a User
- Modify File By many Users
- Get File Latest version
- Get File from last pulled version (incremental synchronization)
- Subscribe to get nofications if the failed gets modified with your own protocol: SNS(email, phone, push notifications, chatbot for Slack), SQS, Kinesis, EventBridge, Kafka, API Gateway and other external systems
- Delete File by the Same User which inserted the file
- Notify Users through SNS or other Subscribing technologies like AppSync, EventBridge, SQS or other external systems, mail, phone, push notifications, frontend apps. For example, I connected successfully with SNS Email and connected my mail and I know get notifications when a file is updated in real time.

Initial Overview Architecture



Event Driven Architecture with AWS Serverless

Event-driven architecture is an architectural pattern that primarily focuses on the production, detection, consumption, and reaction to events that occur within a system. In this architecture, the flow of information is triggered by events rather than being controlled by a central program flow.

In an event-driven architecture, events can be any significant occurrence or change in the system or external environment, such as a user action, system notification, or sensor input. These events are typically asynchronous and decoupled from the components that produce or consume them.

One of the popular implementations of event-driven architecture is through AWS Serverless services. AWS offers various serverless services like AWS Lambda, Amazon EventBridge, and Amazon Simple Notification Service (SNS) that enable the creation and operation of event-driven systems.

By using these AWS serverless services, you can build scalable, event-driven architectures that react to changes and events in real-time, enabling you to create responsive and resilient systems.

Services Used

I used the following AWS services: **API Gateway, Lambda, DynamoDB, SNS, S3, Cognito, SQS, AppSync, EventBridge.** I used these serverless services because: they provide scalability, flexibility, cost-efficiency, and easy integration with other AWS services.

API Gateway is used as the entry point for the system, allowing users to interact with the solution through RESTful APIs. It handles authentication and authorization using Cognito, ensuring only authorized users can access the system.

Lambda functions are used for business logic and processing tasks. They are responsible for handling file insertions, modifications, deletions, and retrieving the latest and last pulled versions of a file.

DynamoDB is used as the main database to store file metadata, such as file names, versions, and user information. It provides fast and scalable storage with automatic scaling based on demand.

S3 is used for storing the actual file content. When a user inserts or modifies a file, the content is stored in S3 and the metadata is stored in DynamoDB. This separation allows for efficient storage and retrieval of file data.

SNS is used for sending notifications to subscribed users when a file they are interested in gets modified. Subscribers can choose to receive notifications through email, SMS, or other supported protocols.

Overall, this solution provides a scalable and reliable way to synchronize craft data across multiple users. It ensures that users can insert, modify, and delete files while maintaining the latest and last pulled versions. Subscriptions and notifications also enable users to stay updated on changes to the files they are interested in.

Implemented Solution

The implemented solution is not finished.

The services used are: API Gateway, Lambda, DynamoDB, SNS. SNS will be used to send the notifications.

To complete the solution, the remaining AWS services to be utilized are S3, Cognito, SQS, AppSync, and EventBridge:

S3 will be used to store the actual file content. Whenever a user inserts or modifies a file, the content will be stored in S3. This separation of content and metadata ensures efficient storage and retrieval of file data.

Cognito will handle user authentication and authorization. It ensures that only authorized users can access the system through API Gateway. This adds an extra layer of security to the solution.

The following services will be used in order to give the customers the opportunity to have more ways to get notified that a file that they suscribed to.

SQS (Simple Queue Service) will be used to handle asynchronous processing of tasks, particularly file modifications. When a file is modified, a message will be sent to an SQS queue of customer.

AppSync will be used for real-time data synchronization between clients and the server. This service enables clients to subscribe to file updates and receive real-time updates whenever a file is modified by other users. It ensures that users are always working with the latest version of a file and can collaborate in real-time. It works nicely with frontend clients.

EventBridge will be used for managing and routing events within the system. It facilitates an event-driven architecture, allowing different components of the solution to react to specific events and trigger corresponding actions. This enables seamless integration and coordination between different services.

With the integration of these services, the Craft Data Synchronization Solution with AWS will provide a comprehensive and efficient platform for users to insert, modify, delete, and retrieve files while maintaining the latest and last pulled versions. Subscriptions and notifications will keep users updated on changes to the files they are interested in. The solution will be scalable, reliable, and cost-efficient, leveraging the power of serverless services and the flexibility of AWS.

Infrastructure

For provisioning the infrastructure I used AWS Cloudformation and AWS CDK.

I chose AWS Cloudformation and AWS CDK for infrastructure provisioning due to their ease of use and ability to automate the process. By using these tools, I was able to define my infrastructure as code, allowing for version control and easy replication.

With AWS Cloudformation, I was able to define all the necessary resources and configurations in a simple YAML or JSON template. This template acted as a blueprint for my infrastructure, specifying everything from EC2 instances and load balancers to security groups and IAM roles. I could easily define the desired state of my infrastructure and let Cloudformation handle the provisioning and management.

However, I also wanted to take advantage of the benefits of programming languages to define more complex infrastructure configurations. This is where AWS CDK came in handy. With CDK, I could write my infrastructure as code using familiar programming languages like Python, TypeScript, or Java.

CDK offered a higher level of abstraction, making it easier to write reusable components and define infrastructure patterns. I could define constructs that encapsulated common infrastructure patterns or resources, and then use them across different projects or environments.

Another advantage of using CDK was the ability to leverage the power of AWS SDKs and services directly within my infrastructure code. This meant that I could use the full range of AWS services, such as AWS Lambda functions or Amazon DynamoDB databases, without having to rely on custom resources or workarounds.

The combination of AWS Cloudformation and CDK allowed me to easily provision and manage my infrastructure in a consistent and repeatable manner. With a single command, I could deploy my entire infrastructure stack, including all resources and configurations.

In conclusion, by using AWS Cloudformation and AWS CDK, I was able to provision my infrastructure efficiently and effectively. These tools provided me with the flexibility, scalability, and automation needed to meet the demands of my project.

Improvements

An improvement would be that the customers would write directly to S3 and bypassing the limits of API Gateway and Lambda. We can even use multi-part upload for files even bigger than 5 GBs and only specify the key of the huge object and store it in the Metadata Files Table. Also, we can leverage the S3 versioning bucket system.

Additionally, implementing a caching layer using services like Amazon CloudFront or Amazon ElastiCache can help improve the performance and reduce the latency of retrieving craft data. This can be particularly beneficial when dealing with large files or high traffic volumes.

Furthermore, implementing a comprehensive backup and restore strategy is essential to ensure data durability and recoverability. Regularly backing up craft data to a separate storage system, such as Amazon Glacier, and defining a robust data retention policy can protect against data loss and enable easy restoration if needed.

Lastly, implementing a comprehensive monitoring and logging solution using services like Amazon CloudWatch and AWS CloudTrail can provide real-time visibility into the system's performance, usage, and potential issues. This allows for proactive identification and resolution of any operational or performance-related issues.

By incorporating these improvements, the Craft Data Synchronization Solution with AWS can provide an even more robust, scalable, and reliable platform for users to effectively manage and synchronize craft data from various sources.

Limitations

Current limitations are the limitations of Lambda, API Gateway, DynamoDB.

DynamoDB, as a NoSQL database service, has its own set of limitations as well. One constraint is the maximum item size, which is 400 KB. If an item exceeds this size, it needs to be split into multiple items. DynamoDB also enforces limits on read and write capacity units, which determine the throughput capacity of the database. These limits are based on provisioned capacity and can be increased by adjusting the settings or using on-demand capacity.

API Gateway also has its own limitations. For instance, there are limits on the number of requests per second (RPS) that an API Gateway stage can handle. This limit varies based on the account level and region, but it is important to consider these limits when designing highly scalable applications. API Gateway also imposes constraints on payload size, with a maximum limit of 10 MB for both request and response payloads.

Because of the current limitations, the files version cannot be bigger than 400KB (DynamoDB Quota). This is why the desired DataStore would be S3. And also to upload directly to S3 to bypass the API Gateway limitation of max 10MB per request. Also, reading from an S3 secured signed link would make the reading on the client side able to scale.

API Documentation

Current endpoints:

- POST /files - a user can create a new file on which other users can modify and subscribe to see the changes on this file

- POST /files/{fileId}/subscribers - other users can subscribe to an already created file by other users and get Notifications on SNS(email,phone,PUSH Notifications). The user must include the protocol details which will be used to get notified. For instance for protocol, SNS(email), you should provide your email. In the future, it could be extended and added other protocols such as: AWS AppSync, AWS SQS, AWS Kinesis, AWS API Gateway endpoint, Kafka, RabbitMQ or other external services.
- GET /files/{fileId} - get the latest version of the file
- GET /files{fileId}/{version} - get from latest versions from the point where you last synced. For example, I had the version 3, and get file was updated to version 5, I will get only the changes from versions 4 and 5. In this way, I implement the service supports **incremental synchronization** meaning that clients can retrieve only the data records which were updated since the **previous sync operation**.
- PUT /files/{fileId} - a user can modify the file and a new version will be created and the subscribed users will be notified
- DELETE /files/{fileId} - not implemented, the user which created the could delete the file that he created

I exported a Postman collection which test the endpoints(without delete):

```
{
    "info": {
        "_postman_id": "908fc2a8-a264-4540-b257-e31f95981f2f",
        "name": "Realtime craft",
        "schema": "https://schema.getpostman.com/json/collection/v2.1.0/
collection.json"
    },
    "item": [
        {
            "name": "POST  File",
            "request": {
                "method": "POST",
                "header": [],
                "body": {
                    "mode": "raw",
                    "raw": "{\r\n    \"fileId\": \"MyFile2\",\r\n    \"userId\":
\"alex\",\r\n    \"data\": \"This is the beginning of this MyFile2.\"\r\n}",
                    "options": {
                        "raw": {
                            "language": "json"
                        }
                    }
                },
```