Events & Worker Queues

Software Architecture

March 27, 2023

Brae Webb

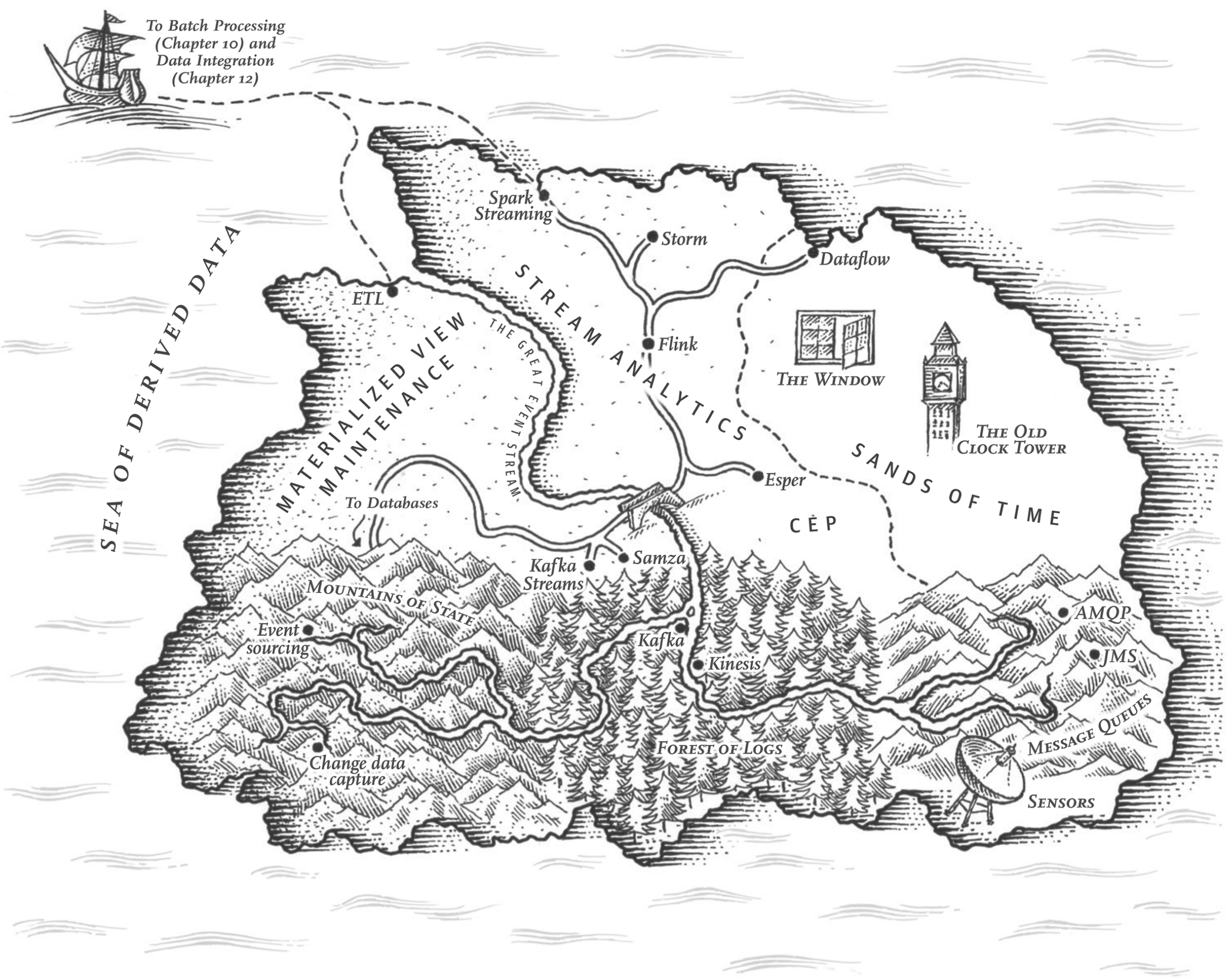


Figure 1: A map of stream processing [1].

1 This Week

This week our goal is to:

- run an application that processes events asynchronously using SQS.

2 Event Processing

As we saw in the Event-Driven Architecture notes [2], event processing enable us to build highly scalable and extensible systems. In this weeks practical we will get our hands dirty with event processing using AWS SQS. SQS gives us a service which can act as an event broker.

2.1 Technologies

We should first discuss a few of our technology options for services which can be used in an Event-Driven Architecture.

2.1.1 AWS SQS

AWS provides the Simple Queue Service, SQS, which offers a simple and fully managed message queue service. There are two flavours of SQS to be aware of; the standard message queue, and the FIFO message queue. Standard message queues allow for greater scalability by providing higher through-put. However, standard message queues in SQS are not exactly queues, messages are not first in first out, they are best-effort ordered. The second flavour, FIFO message queues, guarantees that messages are First in First Out.

2.1.2 AWS SNS

> Amazon Simple Notification Service (Amazon SNS) is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication.
>
> The A2A pub/sub functionality provides topics for high-throughput, push-based, many-to-many messaging between distributed systems, microservices, and event-driven serverless applications. Using Amazon SNS topics, your publisher systems can fanout messages to a large number of subscriber systems, including Amazon SQS queues, AWS Lambda functions, HTTPS endpoints, and Amazon Kinesis Data Firehose, for parallel processing. The A2P functionality enables you to send messages to users at scale via SMS, mobile push, and email.
>
> — AWS

2.1.3 AWS MQ / Apache ActiveMQ / RabbitMQ

> Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that makes it easy to set up and operate message brokers on AWS. Amazon MQ reduces your operational responsibilities by managing the provisioning, setup, and maintenance of message brokers for you. Because Amazon MQ connects to your current applications with industry-standard APIs and protocols, you can easily migrate to AWS without having to rewrite code.
>
> — AWS

Aside

Not available in the lab environments

2.1.4 AWS MSK (Managed Streaming for Apache Kafka)

> Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully managed service that enables you to build and run applications that use Apache Kafka to process streaming data. Amazon MSK provides the control-plane operations, such as those for creating, updating, and deleting clusters. It lets you use Apache Kafka data-plane operations, such as those for producing and consuming data It runs open-source versions of Apache Kafka. This means existing applications, tooling, and plugins from partners and the Apache Kafka community are supported without requiring changes to application code. You can use Amazon MSK to create clusters that use any of the Apache Kafka versions listed under Supported Apache Kafka versions.
>
> — AWS

Aside

Not available in the lab environments

2.1.5 Redis

> Redis, which stands for Remote Dictionary Server, is a fast, open source, in-memory, key-value data store. The project started when Salvatore Sanfilippo, the original developer of Redis, wanted to improve the scalability of his Italian startup. From there, he developed Redis, which is now used as a database, cache, message broker, and queue.
>
> Redis delivers sub-millisecond response times, enabling millions of requests per second for real-time applications in industries like gaming, ad-tech, financial services, healthcare, and IoT. Today, Redis is one of the most popular open source engines today, named the "Most Loved" database by Stack Overflow for five consecutive years. Because of its fast performance, Redis is a popular choice for caching, session management, gaming, leaderboards, real-time analytics, geospatial, ride-hailing, chat/messaging, media streaming, and pub/sub apps.
>
> AWS offers two fully managed services to run Redis. Amazon MemoryDB for Redis is a Redis-compatible, durable, in-memory database service that delivers ultra-fast performance. Amazon ElastiCache for Redis is a fully managed caching service that accelerates data access from primary databases and data stores with microsecond latency. Furthermore, ElastiCache also offers support for Memcached, another popular open source caching engine.
>
> — AWS

Aside

Not available in the lab environments

3 Talking to the Simple Queue Service (SQS)

Warning

For terminal examples in this section, lines that begin with a $ indicate a line which you should type while the other lines are example output that you should expect. Not all of the output is captured in the examples to save on space.

Today we will be creating and experimenting with the two queue flavours of AWS SQS. A standard queue, named `csse6400_prac` and a FIFO queue, named `csse6400_prac.fifo`. The Terraform code below can be used to create these two queues.

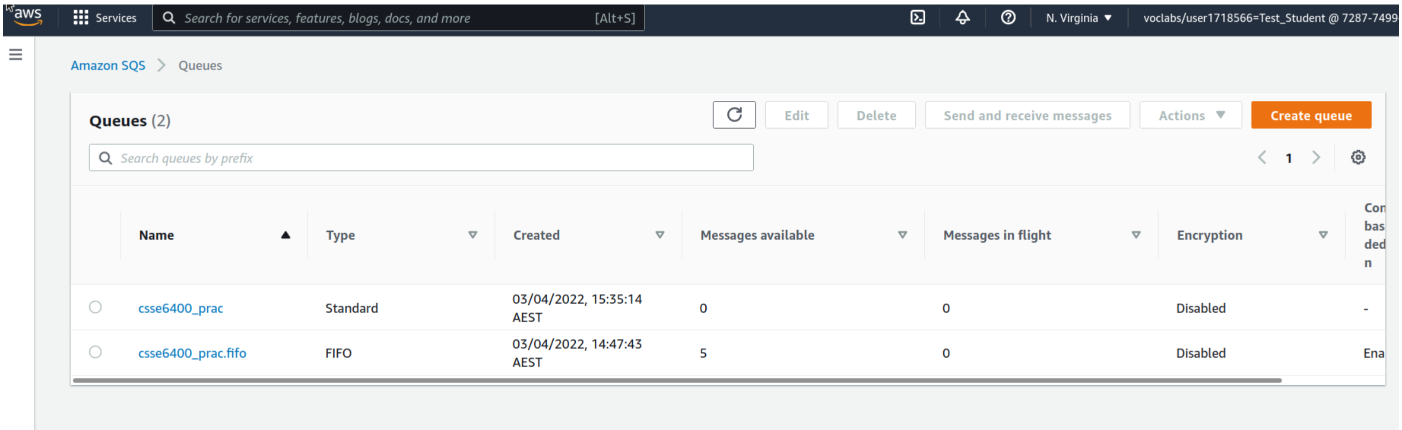
Info

If you have forgotten how to get started you will need to run the following commands in a local terminal.

```
$ terraform init
...
$ terraform plan
...
$ terraform apply
...
```

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_file = "./credentials"
}

resource "aws_sqs_queue" "our_first_mailbox" {
  name = "csse6400_prac"
}

resource "aws_sqs_queue" "our_first_fifo" {
  name = "csse6400_prac.fifo"
  fifo_queue = true
  content_based_deduplication = true
}

output "mailbox" {
  value = aws_sqs_queue.our_first_mailbox.arn
}

output "fifo" {
  value = aws_sqs_queue.our_first_fifo.arn
}
```

Now that we have provisioned the queues we can have a look at them in the AWS Console. In the main AWS dashboard you can search for “SQS” to find these queues. You should reach a page like this:



Like the EC2 and RDS dashboards, we can browse the queue configurations and metrics.

3.1 Queue Command-line Interface

We have provided a small docker container for you to use with your queues to see the differnce between the implementations. First we must retrieve our AWS credentials and setup our environment.

With our learner lab grab the AWS credentials but instead of creating a credentials file we will be using environment variables. Make a folder for the practi.

```
$ mkdir queues && cd queues
```

Now we need to create an environment file for our docker container to read so that it can access AWS. Create a ".env" file in the current directory and edit the contents so that it looks similar to the below: The AWS keys will be from the credentials shown in your lab environment.

```
TERM=xterm-256color
AWS_ACCESS_KEY_ID=...
AWS_SECRET_ACCESS_KEY=...
AWS_SESSION_TOKEN=...
```

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "test" --
    client-name "Client 1"



  __________
 |   \XX/   |
 | T. \/ .T |      University of Queensland
 | XX:   :XX |         Faculty of EAIT
 T L' /\ 'J T
  \  /XX\  /        CSSE6400 Queue Prac
@\_ '____' _/@      csse6400.uqcloud.net
\_X\_ __ _/X_/
 \=/\----/\=/
```

```
Unable to find a Queue by this name test
```

If your program shows the above then your ready to head to the next section :).

3.2 SQS Standard

As we have stated above the “Standard” offering of SQS does not guarantee order or “only once delivery”. Assuming our terraform was created sucessfully we are going to create one publisher and one subscriber.

> **Info**
>
> For the rest of this practical you will require multiple terminals. Make sure these are all in the same folder so we can reuse the .enf file.

To create the subscriber run the following command:

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac" --client-name "Client 1" --receive
```

Now lets start a publisher in another terminal but keep both terminals on your screen if you have space.

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac" --client-name "Client 1"
```

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac" --cl
ient-name "Client 1"

       ----------
      |   \XX/   |
      | T. \/ .T |      University of Queensland
      | XX:  :XX |          Faculty of EAIT
      T L' /\ 'J T
       \  /XX\  /         CSSE6400 Queue Prac
     @\_ '____' _/@       csse6400.uqcloud.net
     \_X\_ __ _/X_/
      \=/\----/\=/


Connected to csse6400_prac
Sending Messages:
Working...                                          25% 0:00:19
```

When the publisher connects to the Queue it is going to put 100 messages of increasing increment into the queue. On the subscriber we will be able to see the messages being received, an example is provided below:

```
      | T. \/ .T |      University of Queensland
      | XX:  :XX |          Faculty of EAIT
      T L' /\ 'J T
       \  /XX\  /         CSSE6400 Queue Prac
     @\_ '____' _/@       csse6400.uqcloud.net
     \_X\_ __ _/X_/
      \=/\----/\=/


Connected to csse6400_prac
[11:41:41] Client 1: Message 0                                                   main.py:62
           Client 1: Message 1                                                   main.py:62
[11:41:42] Client 1: Message 2                                                   main.py:62
           Client 1: Message 5                                                   main.py:62
           Client 1: Message 6                                                   main.py:62
[11:41:43] Client 1: Message 8                                                   main.py:62
           Client 1: Message 9                                                   main.py:62
[11:41:44] Client 1: Message 12                                                  main.py:62
           Client 1: Message 13                                                  main.py:62
           Client 1: Message 14                                                  main.py:62
[11:41:46] Client 1: Message 16                                                  main.py:62
```

hopefully like our example you can see that some of the messages arrive out of order. Next add more publishers and subscribers and experiment with the differnt configurations.

> **Info**
>
> When making multiple publishes you may want to change the client-name cli parameter so you can keep track of when the messages arrived at the subscribers.

3.3 SQS FIFO

Now we will experiment with the FIFO based service offered by SQS. Like before we will start a subscriber but make sure the name of the queue matches the FIFO queue we created in terraform.

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac.fifo" --client-name "Client 1" --receive
```

Now lets start a publisher in another terminal but keep both terminals on your screen if you have space.

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac.fifo" --client-name "Client 1"
```

```
^[[D$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac.f
ifo" --client-name "Client 2"


      ----------
     |   \XX/   |
     | T. \/ .T |      University of Queensland
     | XX:  :XX |          Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /         CSSE6400 Queue Prac
    @\_ '____' _/@       csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/


Connected to csse6400_prac.fifo
Sending Messages:
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
$
```

On the subscriber we now see the messages arriving in order which is to be expected.

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac.fifo"
 --client-name "hello" --receive


     __________
    |   \XX/   |
    | T. \/ .T |        University of Queensland
    | XX:  :XX |            Faculty of EAIT
    T L' /\ 'J T
     \  /XX\  /           CSSE6400 Queue Prac
   @\_ '____' _/@         csse6400.uqcloud.net
   \_X\_ __ _/X_/
    \=/\----/\=/


Connected to csse6400_prac.fifo
[11:57:28] Client 1: Message 0                                                    main.py:62
[11:57:29] Client 1: Message 1                                                    main.py:62
           Client 1: Message 2                                                    main.py:62
[11:57:30] Client 1: Message 3                                                    main.py:62
           Client 1: Message 4                                                    main.py:62
```

If we re-run the publisher though we may not see any new messages make it to the consumer. This is because we have asked AWS to dedup messages where it can.

Again as before try experimenting with diffrent publisher / subscriber configurations to see how they behave.

> **Info**
>
> When making multiple publishes you may want to change the client-name cli parameter so you can keep track of when the messages arrived at the subscribers.

> **Warning**
>
> **Please remember to terraform destroy to delete your resources**

4 Worker Queues

One good usecase of queues is for distributing work over many machines to scale to demand. In this exercise we encourage you to have a look at these widly used libraries to see how you could integrate them into a distributed system.

- Python: Celery
- Java: RabbitMQ

The two above libraries are integrated into many of the popular application frameworks aswell. Today we will be using the python library Celery to create a simple worker queue for our Todo App.

4.1 Celery

Celery is a python library which allows you to create a worker queue. It is very popular and is used in many large scale applications. It is also very easy to use and has a lot of documentation.

4.2 Todo App

Our Todo App will use celery to distribute the work of generating a calendar view of the todo items. This will be done by a worker which will be started on a seperate instance. The webserver will then send the worker a message to generate the calendar view. The worker will then generate the calendar view and send it back to the webserver. The webserver will then display the calendar view to the user.

This allows us to queue a job when the user makes changes and asynchronously generate the calendar view. This means that the user will not have to wait for the calendar view to be generated and the webserver will be able to handle other requests.

TODO: Add diagram

TODO: Add worker cli startup option

TODO: Add celery

TODO: Add job request

4.2.1 Design Challenge

Say we observe a user pattern that the majority of users will mark off multiple todo items at once. We have also moved our calendar from a PDF being displayed to the user to an iCal so that its visible in their chosen calendar application.

How could we improve the efficiency of our system to reduce the amount of time it takes to generate the calendar view?

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.

[2] R. Thomas, "Event-driven architecture," April 2022. https://csse6400.uqcloud.net/handouts/event.pdf.