Lesson 13 SQS & SNS

Michael Yang

What is SNS and SQS

- Amazon Simple Queue Service (Amazon SQS) lets you send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

Use cases

Increase application reliability and scale

Amazon SQS provides a simple and reliable way for customers to decouple and connect components (microservices) together using queues.

Decouple microservices and process event-driven applications

Separate frontend from backend systems, such as in a banking application. Customers immediately get a response, but the bill payments are processed in the background.

Ensure work is completed cost-effectively and on time

Place work in a single queue where multiple workers in an autoscale group scale up and down based on workload and latency requirements.

Maintain message ordering with deduplication

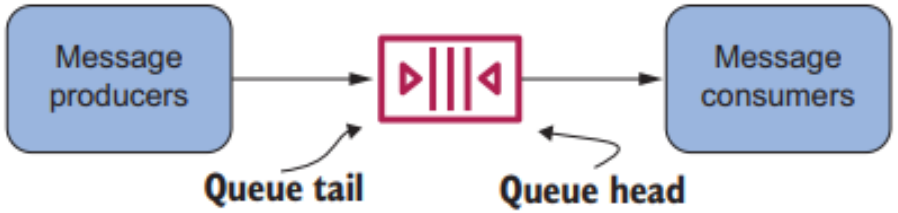
Process messages at high scale while maintaining the message order, allowing you to deduplicate messages.

Asynchronous decoupling

- Why would you want to decouple producers from consumers?

***The queue acts as a buffer*.** Producers and consumers don't have to run at the same speed. For example, you can add a batch of 1,000 messages in one minute while your consumers always process 10 messages per second. Sooner or later, the consumers will catch up, and the queue will be empty again.

***The queue hides your backend*.** Similar to the load balancer, message producers have no knowledge of the consumers. You can even stop all consumers and still produce messages. This is handy while doing maintenance on your consumers.



Producers send messages to a message queue while consumers read messages.

SQS

SQS offers simple but highly scalable—throughput and storage—message queues that guarantee the delivery of messages at least once with the following characteristics:

- Under rare circumstances, a single message will be available for consumption twice. (This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later) .
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced. Learn more about the message order at the end of this section.

Pricing

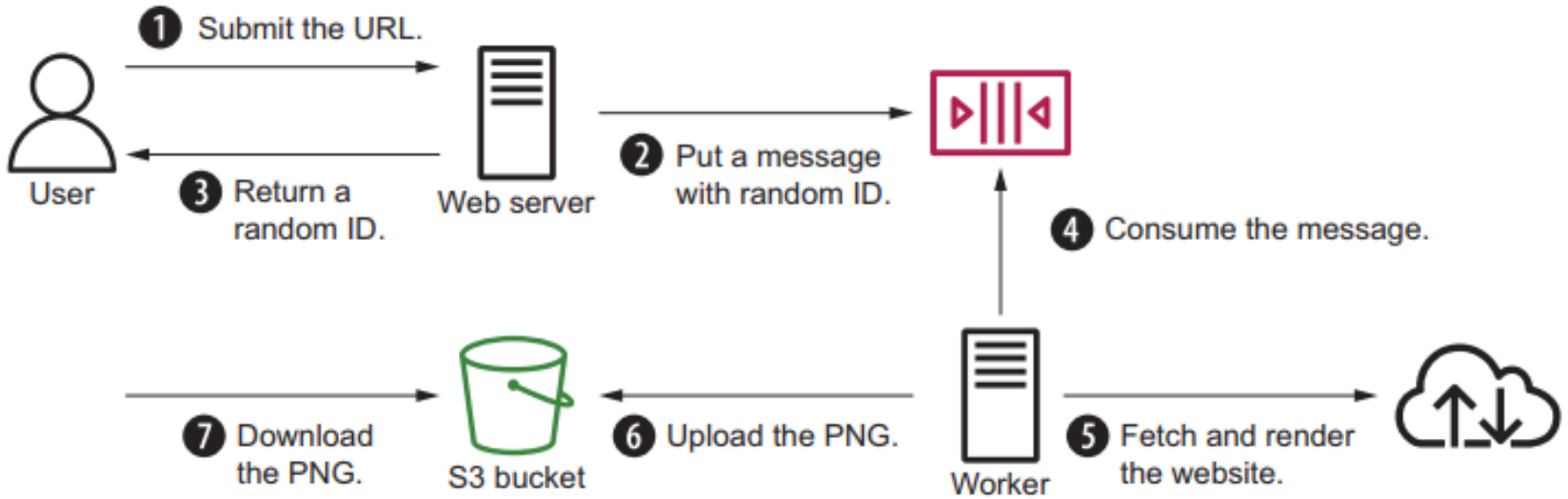
- The pricing model is simple: $0.24 to $0.40 USD per million requests. Also, the first million requests per month are free. It is important to know that producing a message counts as a request, and consuming is another request. If your payload is larger than 64 KB, every 64 KB chunk counts as one request.

Use Case – Synchronous Process

- A typical synchronous process looks like this: a user makes a request to your web server, something happens on the web server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of a URL in the following example, illustrated in figure

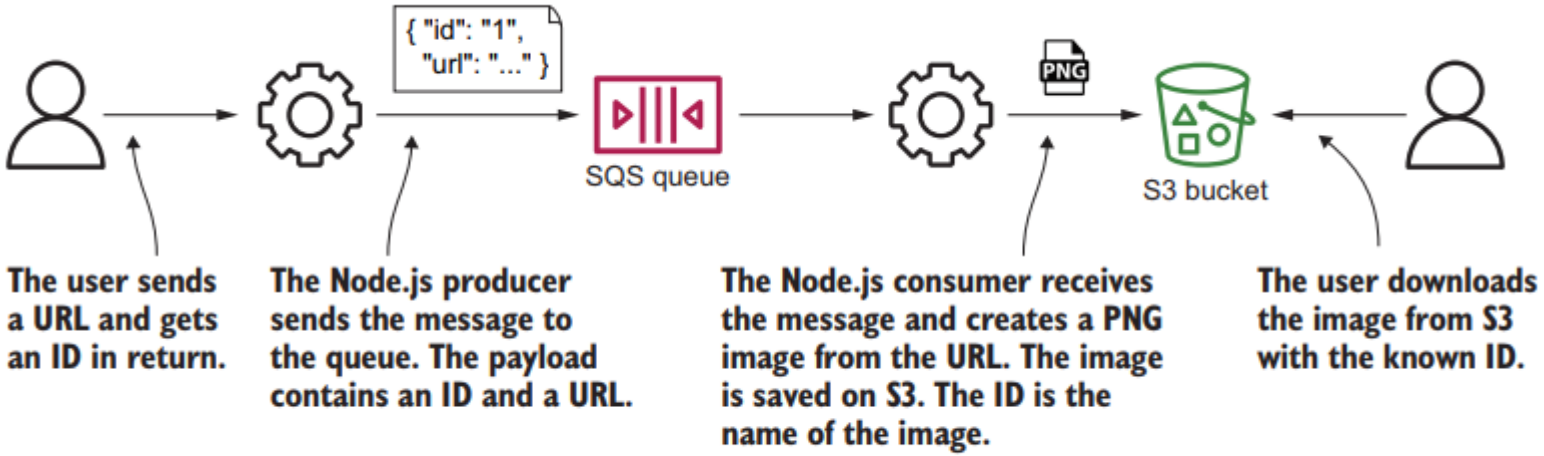
A synchronous process to create a screenshot of a website.

Use Case – Asynchronous Process



The same process, but asynchronous

Programmatically Sending/Receiving Messages



Node.js producer sends a message to the queue. The payload contains an ID and URL.

Producing Messages Programmatically

- You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll use Node.js in combination with the AWS SDK to make requests to AWS.
- Here's how the message is produced with the help of the AWS SDK for Node.js; it will be consumed later by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):
-

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$yourname.s3.amazonaws.com/XYZ.png
```

Producer

```
const AWS = require('aws-sdk');
var { v4: uuidv4 } = require('uuid');
const config = require('./config.json');
const sqs = new AWS.SQS({});

if (process.argv.length !== 3) {
  console.log('URL missing');
  process.exit(1);
}

const id = uuidv4();
const body = {
  id: id,
  url: process.argv[2]
};

sqs.sendMessage({
  MessageBody: JSON.stringify(body),
  QueueUrl: config.QueueUrl
}, (err) => {
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be soon available at http://' + config.Bucket
    ➥ + '.s3.amazonaws.com/' + id + '.png');
  }
});
```

Creates an SQS client

Checks whether a URL was provided

Creates a random ID

The payload contains the random ID and the URL.

Invokes the sendMessage operation on SQS

Converts the payload into a JSON string

Queue to which the message is sent (was returned when creating the queue)

Receiver

- Processing a message with SQS takes the next three steps:
  1 Receive a message.
  2 Process the message.
  3 Acknowledge that the message was successfully processed.

Receive Messages – worker.js

```
const fs = require('fs');
const AWS = require('aws-sdk');
const puppeteer = require('puppeteer');
const config = require('./config.json');
const sqs = new AWS.SQS();
const s3 = new AWS.S3();

async function receive() {
  const result = await sqs.receiveMessage({
    QueueUrl: config.QueueUrl,
    MaxNumberOfMessages: 1,
    VisibilityTimeout: 120,
    WaitTimeSeconds: 10
  }).promise();

  if (result.Messages) {
    return result.Messages[0]
  } else {
    return null;
  }
};
```

Invokes the receiveMessage operation on SQS

Consumes no more than one message at once

Takes the message from the queue for 120 seconds

Long poll for 10 seconds to wait for new messages

Checks whether a message is available

Gets the one and only message

Processing Messages – worker.js

```
async function process(message) {
  const body = JSON.parse(message.Body);
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  await page.goto(body.url);
  page.setViewport({ width: 1024, height: 768})
  const screenshot = await page.screenshot();

  await s3.upload({
    Bucket: config.Bucket,
    Key: `${body.id}.png`,
    Body: screenshot,
    ContentType: 'image/png',
    ACL: 'public-read',
  }).promise();

  await browser.close();
};
```

The message body is a JSON string. You convert it back into a JavaScript object.

Launches a headless browser

Takes a screenshot

The S3 bucket to which to upload the image

The key, consisting of the random ID generated by the client and included in the SQS message

Sets the content type to make sure browsers are showing the image correctly

Allows anyone to read the image from S3 (public access)

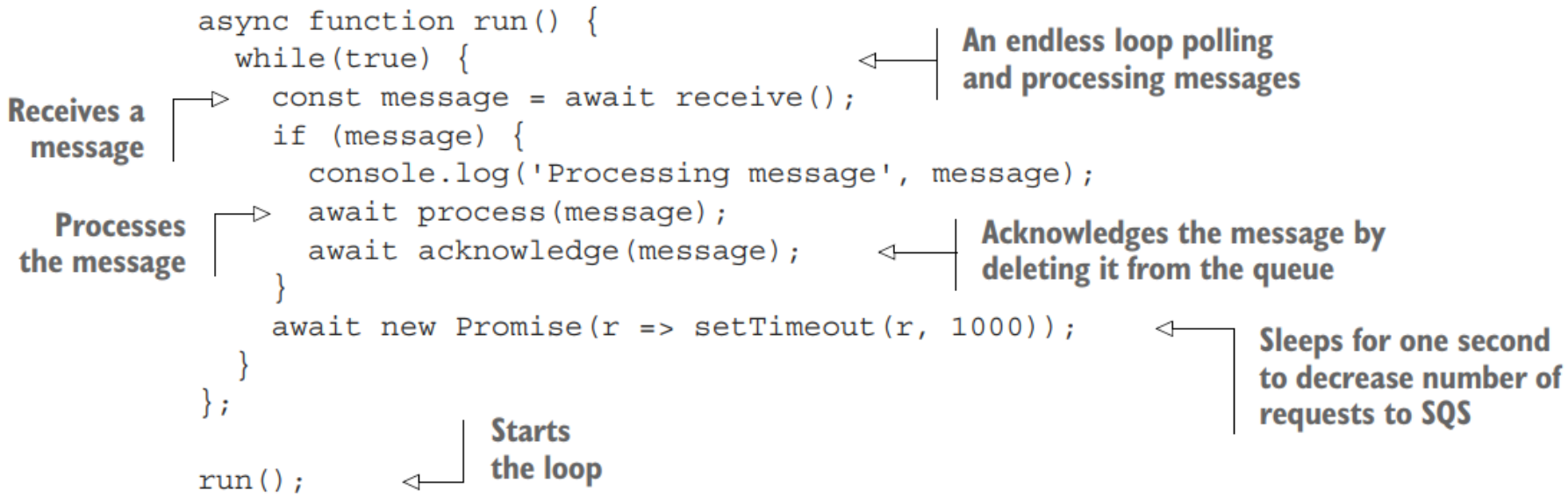
Acknowledging message – worker.js

```
async function acknowledge(message) {
  await sqs.deleteMessage({
    QueueUrl: config.QueueUrl,
    ReceiptHandle: message.ReceiptHandle
  }).promise();
};
```

Invokes the deleteMessage operation on SQS

ReceiptHandle is unique for each receipt of a message.

Run the receiver – worker.js



SQS Benefits

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail. But before we start with the limitations, the benefits include these:

- You can put as many messages into SQS as you like. SQS scales the underlying infrastructure for you.
- SQS is highly available by default.
- You pay per message.

Those benefits come with some tradeoffs. Let's have a look at those limitations in more detail now.

SQS Limitations

- SQS doesn’t guarantee that a message is delivered only once
- SQS doesn’t guarantee the message order
- SQS doesn’t replace a message borker