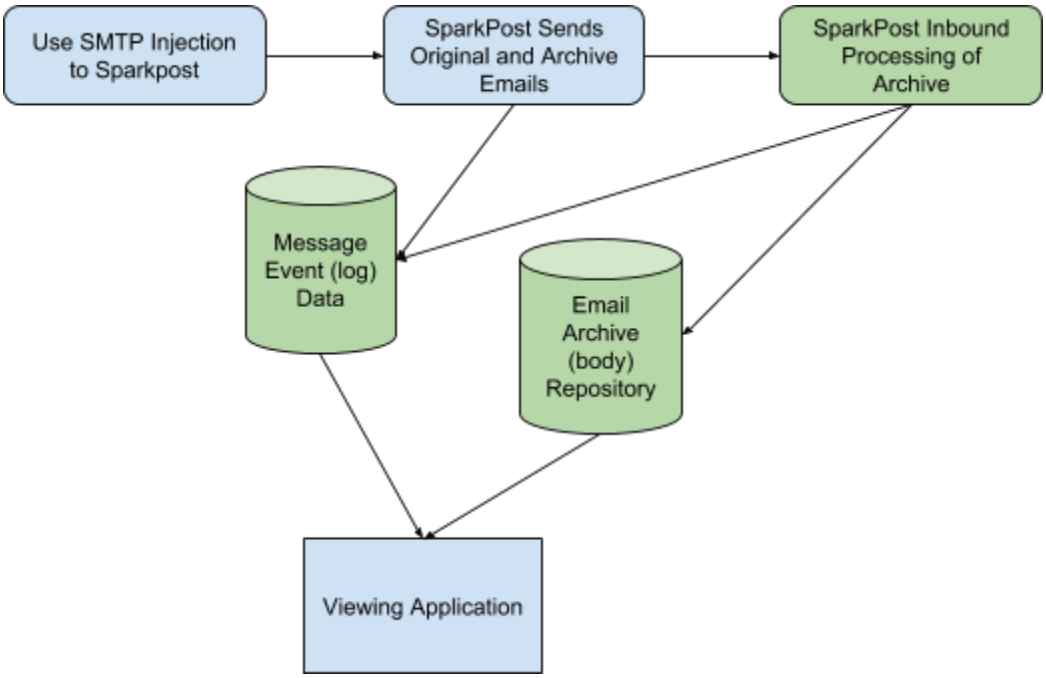
Building an Email Archiving System

Storing the Email Body



* This blog is addressing the process(s) in green

In this blog, I will describe the process I went through to store the body of the email onto S3 (Amazon's Simple Store Service) and ancillary data into a MySQL table for easy cross-referencing. Ultimately, this is the starting point for the code base that will include an application that will allow for easy searching of archived emails, then displaying those emails along with the event (log) data. The code for this project can be found in the following GitHub repository: https://github.com/jeff-goldstein/PHPArchivePlatform.

While I will leverage S3 and MySQL in this project, by no means are these the only technologies that can be used to build an archiving platform, but given their ubiquitousness, I thought they were a good choice for this project. In a full-scale high volume system I would use a higher performance database than MySQL, but for this sample project MySQL is perfect.

I have detailed below, the steps taken in this first phase of the project.

1. Creating the duplicate email for archiving
2. Use SparkPost's Archiving and Inbound Relay features to send a copy of the original email back to SparkPost for processing into a JSON structure then sent to a webhook collector (application)
3. Dismantle the JSON structure to obtain the components necessary
4. Send the body of the email to S3 for storage
5. Log an entry into MySQL for each email for cross referencing

Creating a duplicate of the email

In SparkPost the best way to archive an email is to create an identical copy of the email specifically designed for archival purposes. This is done by using SparkPost's *Archive* feature. SparkPost's *Archive* feature gives the sender the ability to send a duplicate of the email to one or more email address. This duplicate uses the same tracking and open links as the original. The SparkPost documentation defines the Archive feature in the following way:

> *Recipients in the archive list will receive an exact replica of the message that was sent to the RCPT TO address. In particular, any encoded links intended for the RCPT TO recipient will be identical in the archive messages*

The only difference between this archive copy and the original RCPT TO email is that some of the headers will be different since the target address for the archiving email is different, but the body of the email will be an exact replica!

If you want a deeper explanation, here is a link to the SparkPost documentation on creating duplicate (or *archive*) copies of an email. Sample X-MSYS-API headers for this project are shown later in this blog.

There is one caveat to this approach; while all of the event information in the original email is tied together by both a *transmission_id* and a *message_id*, there is no information in the *inbound relay event* (the mechanism for obtaining and disseminating the archive email) for the duplicate email that ties back to one of those two id's and thus the information for the original email. This means we need to place data in the email body and the header of the original email as a way to tie together all of the SparkPost data from the original and archive email.

In order to create the code that is placed into the email body, I used the following process in the email creation application.

1. Somewhere in the email body I placed the following input entry:

```
<input name="ArchiveCode" type="hidden" value="<<UID>>">
```

2. Then I created a unique code and replaced the <<UID>> field:

    ```
    $uid = md5(uniqid(rand(), true));
    $emailBody = str_replace("<<UID>>,$uid,$emailBody);
    ```

    Here is an example output:

    ```
    <input name="ArchiveCode" type="hidden" value="00006365263145">
    ```

3. Next I made sure I added the $UID to the meta_data block of the X-MSYS-API header. This step makes sure that the UID is embedded into each event output for the original email:

```
X-MSYS-API: {
      "campaign_id": "<my_campaign>",
      "metadata" : {
            "UID": "<UID>"
      },
      "archive": [
            { "email": "archive@geekwithapersonality.com" }
      ],
      "options" : {
            "open_tracking": false,
            "click_tracking": false,
            "transactional": false,
            "ip_pool": "<my_ip_pool>"
      }
}
```

Now we have a way to tie all of the data from the original email to the email body of the archive.

Obtaining the Archive version

In order to obtain a copy of an email for archive, you need to take the following steps:

1. Create a subdomain that you will send all archive (duplicate) email(s) to
2. Set the appropriate DNS records to have all emails sent to that subdomain to SparkPost
3. Create an inbound domain in SparkPost
4. Create an inbound webhook in SparkPost
5. Create an application (collector) to receive the SparkPost webhook data stream

The following two links can be used to help guide you through this process:

1. SparkPost technical doc: Enabling Inbound Email Relaying & Relay Webhooks
2. Also, the blog I wrote last year, Archiving Emails: A How-To Guide for Tracking Sent Mail will walk you through the creation of the inbound relay within SparkPost

* Note; as of Oct 2018, the *Archive* feature only works when sending emails using an SMTP connection to SparkPost, the RESTful API does not support this feature. That probably isn't an issue because most emails that need this level of audit control tend to be personalized emails that are fully built out by a backend application before email delivery is needed.

Obtaining the duplicate email in a JSON structure

In the first phase of this project, all I'm storing is the rfc822 email format in S3 and some high-level description fields into a SQL table for searching. Since SparkPost will send the email data in a JSON structure to my archiving platform via webhook data streams, I built an application (often referred to as a *collector*) that accepts the ***Relay_Webhook*** data stream.

Each package from the SparkPost Relay_Webhook will contain the information of one duplicate email at a time, so breaking the JSON structure down into the targeted components for this project is rather straightforward. In my PHP code, getting the rfc822 formatted email was as easy as the following few lines of code:

```
if ($verb == "POST") {
    $body = file_get_contents("php://input");
    $fields = json_decode($body, true);
    $rfc822body =
$fields['0']['msys']['relay_message']['content']['email_rfc822'
];
    $htmlbody =
$fields['0']['msys']['relay_message']['content'][html']
    $headers =
$fields['0']['msys']['relay_message']['content']['headers'];}
```

Some of the information that I want to store into my SQL table reside in an array of header fields. So I wrote a small function that accepted the header array and looped through the array in order to obtain the data I was interested in storing:

```
function get_important_headers($headers, &$original_to,
&$headerDate, &$subject, &$from)
{
    foreach ($headers as $key => $value) {
        foreach ($value as $key_sub => $value_sub) {
```

```
                if ($key_sub == 'To') $original_to = $value_sub;
                if ($key_sub == 'Date') $headerDate = $value_sub;
                if ($key_sub == 'Subject') $subject = $value_sub;
                if ($key_sub == 'From') $from = $value_sub;
            }
        }
    }
```

Now that I have the data, I am ready to store the body into S3.

Storing the duplicate email in S3

I'm sorry to disappoint you but I'm not going to give a step by step tutorial on creating an S3 bucket for storing the email nor am I going to describe how to create the necessary access key you will need in your application for uploading content to your bucket; there are better tutorials on this subject than I could ever write. Here a couple of articles that may help:

https://docs.aws.amazon.com/quickstarts/latest/s3backup/step-1-create-bucket.html
https://aws.amazon.com/blogs/security/wheres-my-secret-access-key/

What I will do is to point out some of the settings that I chose that pertain to a project like this.

1. **Access Control**. You not only need to set the security for the bucket, but you need to set the permissions for the items themselves. In my project, I use a very open policy of *public-read* because the sample data is not personal and I wanted easy access to the data. You will probably want a much stricter set of ACL policies. Here is a nice article on ACL settings:

   https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html

2. **Archiving the Archive**. In S3 there is something called Lifecycle Management. This allows you to move data from one type of S3 storage class to another. The different storage classes represent the amount of access you need to the stored data with lower costs associated with the storage you access the least. A good write up of the different classes and transitioning through them can be found in an AWS guide called, Transitioning Objects. In my case, I chose to create a lifecycle that moved each object from Standard to Glacier after one year. Glacier access is much cheaper than the standard S3 archive and will save me money in storage costs.

Once I have the S3 bucket created and my settings in place, S3 is ready for me to upload the rfc822 compliant email that I obtained from the SparkPost Relay Webhook data stream. But

before uploading the rfc822 email payload to S3 I need to create a unique filename that I will use to store that email.

For the unique filename, I'm going to search the email body for the hidden id that the sending application placed into the email and use that id as the name of the file. There are more elegant ways to pull the connectorId from the html body, but for simplicity and clarity I'm going to use the following code:

```
$start = strpos($htmlbody, $inputField);
$start = strpos($htmlbody, "value=", $start) + 7;
$end = strpos($htmlbody, ">", $start) - 1;
$length = $end - $start;
$UID = substr($html, $start, $length);
```

* we are assuming that $inputField holds the value "ArchiveCode" and was found in my config.php file.

With the UID, we can then make the filename that will be used in S3:

```
$fileName = $ArchiveDirectory . '/' . $UID . '.eml';
```

Now I'm able to open up my connection to S3 and upload the file. If you look at the s3.php file in the GitHub repository you will see that it takes very little code to upload the file.

My last step is to log this entry into the MYSQL table.

Storing the Meta Data in MySQL

We grabbed all of the data necessary in a previous step, so the step of storage is easy. In this first phase I chose to build a table with the following fields:

- An automated field entry for date/time
- The target email address (RCPT_TO)
- The timestamp from the email DATE header
- The SUBJECT Header
- The FROM email address header
- The directory used in the S3 bucket
- The S3 filename for the archived email

The function named, MySQLLog within the upload.php application file goes through the necessary steps to open the link to MySQL, inject the new row, test the results and close the

link. I do add one other step for good measure and that is to log this data into a text file. Should I do a lot more logging for errors? Yes. But I do want to keep this code lite in order to allow it to run extremely fast. At times this code will be called hundreds of times per minute and needs to be as efficient as possible. In future updates, I will add ancillary code that will process failures and email those failures to an admin for monitoring.

Wrapping it up

So in a few fairly easy steps, we were able to walk through the first phase of building a robust email archiving system that holds the email duplicate in S3 and cross-referencing data in a MySQL table. This will give us a foundation for the rest of the project that will be tackled in several future posts.

In future revisions of this project I would expect to:

1. Store all log events of the original email
2. Send storage errors to an admin when a failure to upload or log happens
3. Minimize the collector complexity.
4. Add a UI for viewing all data
5. Support the ability to resend the email

In the meantime, I hope this project has been interesting and helpful to you; happy sending.