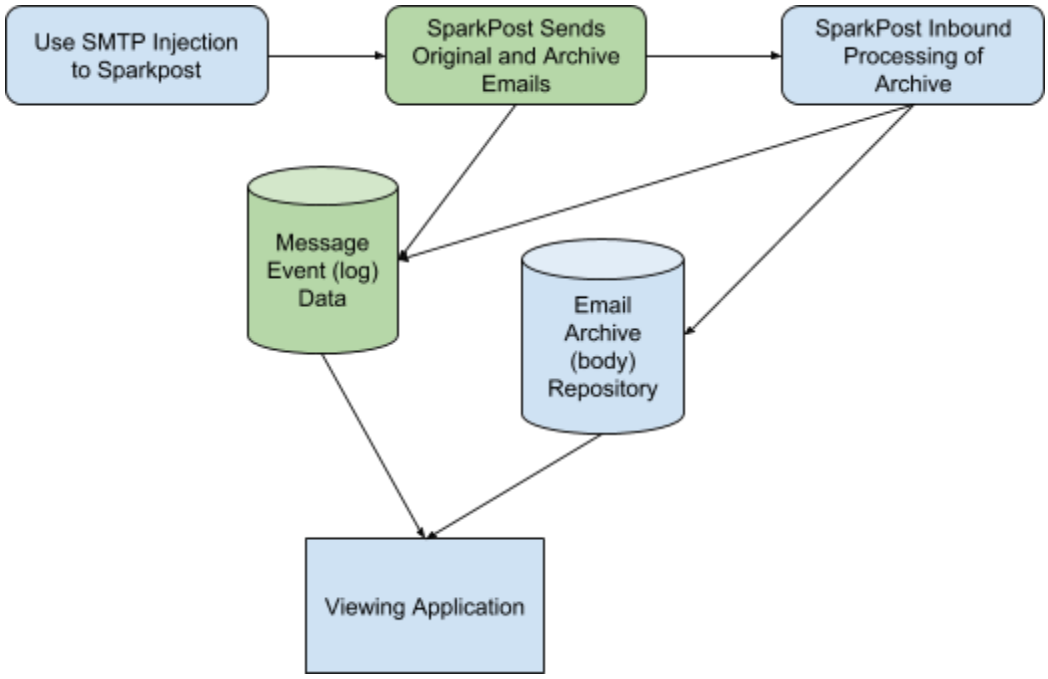
Building an Email Archiving System

Storing the email log data



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

In the third installment of this blog series and the second in the coding phase, I'm diving back into the process of storing the log (event) data associated with the original email. This step will give us the information around the email injection, delivery and behavioral actions by the email recipient(s). While this blog is similar to the blog I wrote recently on storing all event data for an email, there is enough of a twist in order to support the archiving of the email body that I decided to keep the two blogs separate. This blog will also offer up some sample code for storing the data into an MySQL table. (Please refer to the following Github repository)

When an email is created, SparkPost logs each step of the email and makes the data available to you for storage via API or Webhook technologies. Currently, there are 14 different events that may happen to an email. Here is a list of the current events:

Bounce	Click	Delay
Delivery	Generation Failure	Generation Rejection
Initial Open	Injection	Link Unsubscribe
List Unsubscribe	Open	Out of Band
Policy Rejection	Spam Complaint	

* Follow this link for an up to date reference guide for a description of each event along with the data that is shared for each event.

Each log event that corresponds to our archiving project will have a special tag named *uid* within the *metadata* block. As described in the previous two blogs, the *uid* is an application generated field which is generated by your email injection systems during the email send process and placed into the X-MSYS-API header **and** the email body via a hidden html field. The *uid* field in the email body will be the the only id that survives through all emails and logging events and thus is needed to pull everything together (remember that the inbound event data does not have the UID meta data, that is why we must hide the id in the email body). But all of the event log data in this step will have the UID entry that we need.


```
"raw_rcpt_to": "austein@hotmail.com",
"rcpt_meta": {
        "UID": "0093983927113301"
},
"rcpt_tags": [],
"rcpt_to": "austein@hotmail.com",
"rcpt_type": "archive",
"recv_method": "esmtp",
```


SparkPost webhooks have the ability to allow the user to pick which events they want to be delivered to their endpoint (collector), but they don't have the ability filter specific events given specific data. For example, what if you sent welcome emails to your new customers and you only want *open* events with the *subject* line of 'Welcome' sent to a specific endpoint: Nope, can't do that. You can filter on the *click* event, but not on given data within the *subject* field. That means that our endpoint will get all events of a given type (open, click, bounces, etc) and we need to filter out the data that has nothing to do with our archive process. To filter out the noise, we will search each event for the *UID* field within the *metadata* block. If the event has the *UID* field, then we care about that data; otherwise we skip that data event (that also means, that the field name that you use for the *UID* needs to be unique to this project).

* Note: SparkPost does have the ability to filter events for subaccounts. To simplify the storing code, you could send all emails that need to be archived through a specific subaccount. That would allow SparkPost to filter out the events for just that subaccount and only send those events to your collector. It won't save you a lot of code, but this is an option.

In this phase of the code, I have an endpoint that captures and stores all events from SparkPost into a directory. **That is all the work the endpoint does**. This follows the best practice of doing as little work as possible within the endpoint, so it can keep up with how fast SparkPost may be delivering data to that endpoint (this is the *retrieveWebhook.php* code). Being honest, this is NOT the approach I did with the first phase of the project when I stored the archive body. I do plan on going back to fix that at a later date.

The next step will be for a cron job or group of cron jobs that will read the directory of files and start processing them (this is the *processOutboundWebHooks.php* code).

In order to support high volume sending, I expect that this code may have multiple instances running in parallel. So I have the code read the list of files within a directory and try to lock the file it wants to process. If the lock process works, it will proceed; if the lock process doesn't work, then it's assumed that another process is working on that file and skips that file and go onto the next one in the list. Once your code has a file, we need to turn the data into an array and start to process each event separately. But remember, we only want the ones with the uid field; that tells use that this data belongs to the archive process and we may want to store that data. In my code, I loop through each event pulling out specific fields that I know I want to store.

Also for filtering purposes, there is an important key/value pair that we I'm paying special attention to, it's the *rcpt_type* key/value pair. When an email is sent out using either the *cc*, *bcc*, or the *archive* feature, each event will have a corresponding *rcpt_type* of either *'cc'*, *'bcc'* or *'archive'*. My design allows for the email administrator to decide if they want to keep or filter those out by placing the appropriate values in the config file.

The PHP code to decide on if this event should be stored or not looks like this in my project:

```
if ($uid)
{
        switch (true)
        {
                case ($rcpt_type == "original" | $rcpt_type == "archive"):
                        $store = true;
                        break;
                case ($rcpt_type == "cc" && $logCC):
                        $store = true;
                        break;
                case ($rcpt_type == "bcc" && $logBCC):
```

```
                    $store = true;
                    break;
            }
    }
```

By setting the *$store* flag to *true*; the rest of the program will store the corresponding data for that event.

So in short we have gone through the following steps:

1. The collector gets the data from SparkPost and places into a directory (*retrieveWebhook.php*)
2. Another process(s) will read the directory of files saved by the collector and try to lock the file from other processes. If successful, it will continue with that file. If not, it will continue down it's list of files until it runs out or finds a file that needs processing. (processOutboundWebHooks.php[1])
3. Since the file may have many events; we create an array of each event.
4. Looping through each event, we pull data we are interested in. the *rcpt_type* and *uid* are necessary to decide if we care about this event.
5. If we decide to store this event; we proceed with saving the data to our SQL table.

([1] This is a truly bad name, but I'm trying to describe the process that is storing the web hooks data coming from SparkPost outbound versus the archive inbound process)

In this project some of the more significant fields that I decided to use for indexing are: campaign_id, subject, timestamp, template_id and of course the rcpt_to field which holds the target email address.

This leaves me with a table with the following fields:

- Connector Id
- Rcpt To
- Campaign Id
- Subject
- Timestamp
- Template Id
- Raw (a copy of the full data event)

I'm not going to assume that these are the only fields or the best fields for your implementation, but they are what I'm using for this sample project. The code to store our fields in SQL and a text file log is similar to the one in phase 1:

```
	// Create connection
	$conn = mysqli_connect($servername, $username, $password);

	// Check connection
	if (!$conn)
	{
		if ($loggingFlag) $archive_output = sprintf("\n\n>>>>>MySQL connection failed connecting to MYSQL to log S3 entry:%-200s\nTo: %-50s\n From: %-50s\n Subject: %-200s\n InjectionTime: %-42s\n UID: %-38s\n Event Type: %-38s\n ArchiveFileName: %s>>>>>", $conn->error, $rcpt_to, $friendly_from, $subject, $injection_time, $uid, $event_type, $currentFile);
		$deleteFile = false;
	}
	else
	{
		$sql = "INSERT INTO austein_archiver.events (campaign_id, friendly_from, injection_time, rcpt_to, rcpt_type, subject, UID, event_type, raw) VALUES ('" . $campaign_id . "', '" . $friendly_from . "', '" . $injection_time . "', '" . $rcpt_to . "', '" . $rcpt_type . "', '" . $subject . "', '" . $uid . "', '" . $event_type . "', '" . $raw . "')";
		if ($conn->query($sql) === TRUE)
		{
			if ($loggingFlag) $archive_output = sprintf("\n\n>>>>>To: %-50s\n From: %-50s\n Subject: %-200s\n InjectionTime: %-38s\n UID: %-38s\n Event Type: %-38s\n ArchiveFileName: %s>>>>>", $rcpt_to, $friendly_from, $subject, $injection_time, $uid, $event_type, $currentFile);
		}
		else
		{
			if ($loggingFlag) $archive_output = sprintf("\n\n>>>>>MySQL insert failure to MYSQL Event Table:%-200s\nTo: %-50s\n From: %-50s\n Subject: %-200s\n InjectionTime: %-38s\n UID: %-38s\n Event Type: %-38s\n ArchiveFileName: %s>>>>>", $conn->error, $rcpt_to, $friendly_from, $subject, $injection_time, $uid, $event_type, $currentFile);
			$deleteFile = false;
		}
		mysqli_close($conn);
	}
	file_put_contents("eventLog.txt", $archive_output, LOCK_EX | FILE_APPEND);
```

Without dragging this on, that is all we need to review for this portion of the code. As you can see, most of the work was done in the previous steps so all we have to do is retrieve the data, check to make sure it's data we care about, then log it.

The next code drop will have a sample viewer with something similar to an inbox. Until then, it's up to you to come up with some ways to view the data. If you have something to share, I'm sure everyone will be happy to see your work.

Happy Sending.