

Integrating Visual FoxPro and MailChimp – Part 3

Whil Hentzen

We've all written our own email applications. I finally decided to use an outside service to handle my emailing needs. Here's how I used VFP to integrate with the mailing service.

Last issue's article promised to show you how to download your mailing list, unsubscribe users and get stats about your lists, all from Visual FoxPro instead of having to go to the MailChimp website and perform those tasks manually.

I then showed you how to connect to the Web using West-Wind's client tools, how the MailChimp API worked, how to construct the URL needed to perform an export of a list, and how to use VFP to automate that process. Alas, we ran out of room before I could get to the other tasks, so that's the topic of this article.

Types of Methods

Let's first look at the types of things you can do with the MailChimp API. Their documentation breaks out functions into 12 categories. I initially found the list of categories a little daunting, as the list seemed to be in haphazard order, and there weren't any descriptions of the categories themselves, just of the methods available in each category. Let's take a quick look at the more interesting ones, in the order that you'd use them.

Lists Related

These methods enabled you to manage lists, including adding and removing subscribers, exporting portions of a list, and managing the attributes of a list that allow you to filter it into sub-groups (or segments, as MailChimp calls them.)

Campaigns Related

Once you've got a list, you're going to want to use it, specifically by sending email and tracking the results. These processes are called campaigns. The campaign-related methods enable you to create, send, and delete a campaign, and all the myriad middle steps involved in doing so, such as testing and scheduling.

Goal Related

Once you've sent a campaign, you may want to automatically trigger followup actions, such as sending a second email if they visit a specific page on your website or make a request. MailChimp calls these automatically triggered actions 'goals'. These methods enable you to trigger goal event and retrieve goal event data.

VIP Related

You can tag certain list members as 'super duper special', or, as MailChimp refers to them, VIPs. These methods enable you to perform additional actions for your VIP list members, such as adding and deleting VIP status a a list member and retrieving activity by your VIPs.

Conversations Related

When you send out a campaign, some of the list members will reply to your email. You'll also receive auto-reply, on vacation, and bounceback messages. It's hard to sift through the latter to find the former. MailChimp will do the filtering, calling the replies from live humans 'conversations'. These methods enable you to retrieve conversation messages, metadata, and reply to conversations.

Reports Related

Sending email is oh so much more useful if you can garner information about the results. The Reports methods enable you perform tasks such as getting the addresses (1) that complained about a campaign, (2) for which a campaign bounced, (3) that opened an email, (4) that clicked on a URL in an email, and (5) other stats, such as Google Analytics, Twitter mentions, and so on.

Ecomm Related

MailChimp provides ecommerce plugins that connect to third party ecommerce systems, and there are plugins available that have been developed by third parties. MailChimp's Ecomm Related methods enable you to connect to those

systems via those plugins and import and manage orders.

Templates Related

MailChimp also provides the ability to create templates that can be to format emails sent over multiple campaigns; the Templates Related methods enable you to create, delete and otherwise manage those templates.

Helper Related

This category is sort of a misnomer, I feel; it's got a smorgasbord of good, interesting stuff. First, there are methods that enable you to retrieve account information, such as the type of MailChimp plan you've got, payments made, Contact Info, high level account stats, and the like.

Next, there are methods that enable you to retrieve high level campaign and list data, and utilities to convert HTML to text or inlined with CSS. Finally, there's a simple 'ping' method that enables you to verify that the MailChimp API is up, similar to the

```
'DownForEveryone.com'
```

site that helps you determine availability.

Other Categories

Finally, there are a number of administrative categories too, including Users Related (manage the users who can access your MailChimp account), and Folders and Gallery Related (adding and deleting campaign folders and gallery images and folders.)

Manage Subscribers

In my last article, I showed you how to get a download ('export') a copy of your list from the MailChimp site. I also promised to show you how to manage subscribers. But first, let's look at a meta-list method, list.

List All Lists

The 'list' list method (sort of like wait window nowait) delivers a result set comprising of metadata about the lists in your account. All you need to pass to 'list' is your API key.

```
local lcDC, lcAPIkey, lcMethod, lcURL
lcDC = 'us1'
lcAPIkey = 'secret_apikey'
lcMethod = 'list'

lcURL ;
= "https://" ;
+ lcDC ;
+ ".api.mailchimp.com/2.0/lists/" ;
+ lcMethod ;
+ "?apikey=" ;
+ lcAPIkey + "-" + lcDC
```

Next, set up the West-Wind Client Tools.

```
do wwclient
do wwhttp
o=createobject('wwHTTP')
```

Initialize the variables to be passed and returned. Remember, luResult is the entire result set back passed back, not just a boolean or numeric flag.

```
m.liText = 0
m.lcStringReceived = ''
m.luResult = -1
m.lcErrorMsg = ''
```

If you wanted to watch the fun while you were testing, you could include some debugout commands:

```
debugout 'BEFORE'
debugout 'lcErrorMsg:', lcErrorMsg
debugout 'luResult', luResult
debugout 'lcStringReceived:', lcStringReceived
debugout 'liText:', liText
```

Now make the call:

```
m.luResult=o.HTTPGet(lcURL, ;
@m.lcStringReceived, @m.liText)
```

Grab an error string, just in case.

```
lcErrorMsg = o.cErrorMsg
```

And see what showed up afterwards.

```
debugout 'AFTER'
debugout 'lcErrorMsg:', lcErrorMsg
debugout 'luResult', luResult
debugout 'lcStringReceived:', lcStringReceived
debugout 'liText:', liText
```

If everything worked, luResult would look something like this:

```
{"total":1,"data":
[{"id":"secret_list_id","web_id":1111111,"name
":"Test List","date_created":"2014-10-01
23:29:48","email_type_option":false,"use_awseso
mebar":true,"default_from_name":"Whil Hentzen
(Test)","default_from_email":"whil@whilhentzen
.com","default_subject":"","default_language":
"en","list_rating":0,"subscribe_url_short":"ht
tp://eepurl.com/XXXXX","subscribe_url_long"
:"http://hentzenwerke.us3.list-
managel.com/subscribe?
u=secret_apikey&id=secret_list_id","beamer_add
ress":"us3-
bsecret_other_id@inbound.mailchimp.com","visib
ility":"pub","stats":
{"member_count":1,"unsubscribe_count":0,"clean
ed_count":0,"member_count_since_send":1,"unsub
scribe_count_since_send":0,"cleaned_count_sinc
e_send":0,"campaign_count":0,"grouping_count":
0,"group_count":0,"merge_var_count":2,"avg_sub
_rate":15,"avg_unsub_rate":0,"target_sub_rate"
:0,"open_rate":0,"click_rate":0,"date_last_cam
paign":null},"modules":[]}]}
```

This extensive tuple, with many, many key value pairs, starts with the total # of lists, and then for each list, attributes including (but not limited to) the following:

- List ID
- Web ID
- List Name
- Date Created
- Default From Name
- Default From Email
- Default Language
- Subscription URL (short)
- Subscription URL (full)
- Number of Subscribers
- Number of Unsubscribers

The List ID attribute is a very valuable entity, because with it, you can then roll through all of your lists without having to look the IDs up manually.

Manage Subscribers

With a bit of practice in hand, now let's look at managing subscribers.

Unsubscribe

The first scenario that you will likely run into will be when a subscriber emails you and asks you to remove them, instead of them doing it themselves. You could go into your MailChimp dashboard and do it that way, but how much nicer to have a form in your VFP app for subscriber maintenance. Simply find the email address, hit 'Unsub' and let VFP (and West-Wind) connect to the MailChimp site via the API, and take care of it automatically. Here's

Subscribe

Once in a while, you may need to subscribe someone manually. This typically isn't a good idea, because the double-opt-in process described in my first article is designed to ensure that the subscriber really wanted to subscribe. Subscribing an address via the API can be subverted (although the MailChimp people do monitor, and will suspend an account suspected of abuse.) However, if you're adding addresses of your own, for example, for testing or other purposes, this may be perfectly reliable.

Update Subscriber Profile

A subscriber's profile contains several pieces of information that might need to be updated. Just like someone who wants to be unsubscribed but finds it easier to email you instead of doing it themselves, a subscriber may email you to have them fix a typo in their name, change the segments they've flagged, or change the type of email format.

MailChimp Structs and Arrays

As I mentioned in my last article, the MailChimp API is split into two pieces. The Export functions have one syntax, the syntax we're going to use from now on has a slightly different structure, because the parms passed can be more involved, including the potential for passing arrays. The subscribe, unsubscribe and update methods look roughly like this:

- lists/subscribe (string apikey, string id, struct email, struct merge_vars, string email_type, bool double_optin, bool update_existing, bool replace_interests, bool send_welcome)

- lists/unsubscribe (string apikey, string id, struct email, boolean delete_member, boolean send_goodbye, Boolean send_notify)

- lists/update-member (string apikey, string id, struct email, struct merge_vars, string email_type, boolean replace_interests)

You'll see a common parameter in all of these, the 'struct email', which is not something we use in Visual FoxPro. The 'struct' data structure is essentially a key-value pair. The MailChimp documentation provides examples using a variety of languages, but not VFP, and it's not immediately obvious just how to format such a beast. The format for a 'struct' that passes an email address is such:

```
+ "&email[email]" + lcEmail
```

You can subscribe and unsubscribe multiple addresses via sibling methods, batch-subscribe and batch-unsubscribe. In order to do so, you need to pass an array of email address, and that's done like so:

```
+ "&batch[0][email][email]" + lcEmail1 ;  
+ "&batch[1][email][email]" + lcEmail2
```

Obviously, you'd wrap this in a FOR NEXT loop to process an array of email addresses and concatenate the strings.

Building a Subscribe String

The rest of the call to the subscribe and unsubscribe methods are similar to the List method just described. First, initialize the MailChimp API variables for your specific account. This time, include the List ID and the Email to be subscribed.

```
local lcDC,lcAPIkey,lcMethod,lcListID,lcURL
lcDC = 'us1'
lcAPIkey = 'secret_apikey'
lcMethod = 'subscribe'
lcListID = 'secret_listid'
lcEmail = 'email:muscle@softwaremuscle.com'
```

and build the URL to be passed to West-Wind.

```
lcURL ;
= "https://" ;
+ lcDC ;
+ ".api.mailchimp.com/2.0/lists/" ;
+ lcMethod ;
+ "?apikey=" ;
+ lcAPIkey + "-" ;
+ lcDC ;
+ "&id=" ;
+ lcListID ;
+ "&email[email]=" ;
+ lcEmail
+ ""]]"
```

You may want to build this expression, stuff in values, convert it to a text string and paste the entire string into a Web browser manually, just to confirm that you've got everything just right. Doing so would produce a Web page with a confirmation message as shown in **Figure 1**.

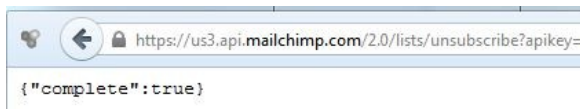


Figure 1. A successful interactive response.

Once you've got the URL formatted and working correctly, go through the same steps with the West-Wind Client Tools setup, initialize the vars to be passed and returned, and make the call.

```
m.luResult=o.HTTPGet(lcURL, ;
@m.lcStringReceived, @m.liText)
```

In this scenario, if everything worked, luResult would look something like this:

```
{\"email\":\"muscle@softwaremuscle.com\",
\"euid\":\"61gq0u7wf7\", \"leid\":\"194560488\"}
```

The euid string is the email unique ID in the system, and the leid string is the unique ID for the

list member in the system. In other words, the email address could be subscribed to multiple lists.

If the call didn't work, however, you'd get an error message that would contain information about what happened. For example, suppose you passed the string

The Quick Brown Fox Jumps

as the list ID parameter of the URL. Well, this isn't a valid ID, so the MailChimp API would respond with the following error:

```
{\"status\":\"error\", \"code\":200, \"name\":\"List_DoesNotExist\", \"error\":\"Invalid MailChimp List ID: The Quick Brown Fox Jumps\"}
```

So that's the basic code. There is more to using the Subscribe code than simply clicking 'Run', though. Let's talk about how to use it.

Once you execute the code, a request with the email address is sent to the MailChimp server, and it's processed just as if someone manually submitted the email address online, as described in the first article in this series.

That means that MailChimp will send an email to the email address being subscribed, as shown in **Figure 2**.

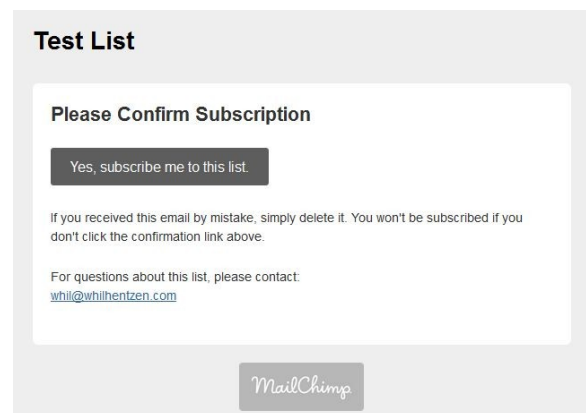


Figure 2. Confirming an automated subscribe request.

This email will have to be responded to, just as if the user had initiated the subscribe process themselves. Once they do, the email address will be added to the list. You can check the stats of your list on the MailChimp site; it'll take a couple of minutes for the subscription to 'hit', but you'll soon see the number of subscribers has increased by 1. See **Figure 3**.

Sort By	Custom Order			
<input type="checkbox"/>	Test List	Created Oct 12, 2014 06:29 pm	1	0.0% 0.0%
		No rating yet	Subscribers	Opens Clicks

Figure 3. An automated subscription request hits!

Building an Unsubscribe String

The basic unsubscribe process is nearly identical. Simply pass 'unsubscribe' instead of 'subscribe' as the value for lcMethod:

```
lcMethod = 'unsubscribe'
```

Now you see why I split the method name out onto a separate line – the method name can be parameterized and the rest of the code that builds the URL string can be executed just once.

```
LcAction = 'sub'
lcAction = 'unsub'

do case
case lcAction = 'sub'
  lcMethod = 'subscribe'
case lcAction = 'unsub'
  lcMethod = 'unsubscribe'
otherwise
  * trap...
endcase
```

So the building of the URL looks like this:

```
lcURL ;
= "https://" ;
+ lcDC ;
+ ".api.mailchimp.com/2.0/lists/" ;
+ lcMethod ;
+ "?apikey=" ;
+ lcAPIkey + "-" + lcDC ;
+ "&id=" + lcListID ;
+ '&email=[{"cemail":"' ;
+ lcEmail ;
+ '"}]]'
```

As mentioned earlier, the list methods encompass far more than just handling subscribers.

Building an Activity String

Aggregate list-related methods don't require an email address. For example, the activity method just needs an API key and a list ID, and will return a tuple of various statistics. The URL would be built like this, where lcMethod = 'activity':

```
lcURL ;
= "https://" ;
+ lcDC ;
+ ".api.mailchimp.com/2.0/lists/" ;
+ lcMethod ;
+ "?apikey=" ;
+ lcAPIkey + "-" + lcDC ;
+ "&id=" + lcListID ;
```

It returns a string like so:

```
[{"user_id":26390867,"day":"2014-10-13","emails_sent":0,"unique_opens":0,"recipient_clicks":0,"hard_bounce":0,"soft_bounce":0,"abuse_reports":0,"subs":1,"unsubs":0,"other_adds":0,"other_removes":0}]
```

Building an Abuse Reports String

Another useful report to pull for a list is the list of email addresses that complained about a campaign sent to that list. Again, the only parameters that must be sent to the method are the API key and the list ID. The URL is identical to the previous statement, except that lcMethod = 'abuse-reports'. The return value looks like this:

```
{"total":0,"data":[]}
```

(Hooray! No complaints!)

Building a List Growth Report string

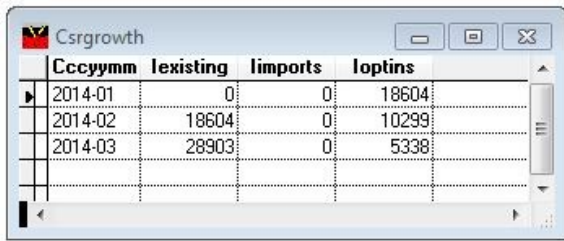
As we're all good at math, we also like numbers and charts. And since we're also interested in growing our lists, the Growth History report is a lot of fun. Once more, the only parameters that must be sent to the method are the API key and the list ID. The URL is identical to the previous statement, except that lcMethod = 'growth-history'. The return value looks like this:

```
[{"month":"2014-01","existing":"0","imports":"0","optins":"18604"}, {"month":"2014-02","existing":"18604","imports":"0","optins":"10299"}, {"month":"2014-03","existing":"28903","imports":"0","optins":"5338"}]
```

Here's a snippet of code to parse your growth history into a cursor:

```
if lcAction = 'grow'
  luResult = strtran(luResult, '[', '')
  luResult = strtran(luResult, ']', '')
  * how many months to parse?
  liHowManyMonths = occurs('month', luResult)
  create cursor csrGrowth (cCCYYMM c(7), ;
    iExisting i, iImports i, iOptins i)
  x = luResult
  for li = 1 to liHowManyMonths
    lcCCYYMM = substr(x, at('month',x,li)+8,7)
    liExisting ;
      = val(substr(x, at('existing',x,li)+11, ;
        (at('imports',x,li)-3) ;
        - (at('existing',x,li)+11)))
    liImports ;
      = val(substr(x, at('imports',x,li)+10, ;
        (at('optins',x,li)-3) ;
        - (at('imports',x,li)+10)))
    liOptins ;
      = val(substr(x, at('optins',x,li)+9, ;
        (at('')',x,li)) ;
        - (at('optins',x,li)+9)))
    insert into csrGrowth ;
      (cCCYYMM,iExisting,iImports,iOptins) ;
      values ;
      (lcCCYYMM,liExisting,liImports,liOptins)
  next
endif
```

See **Figure 4** for an example of the cursor created.



Cccymm	lexisting	limports	loptins
2014-01	0	0	18604
2014-02	18604	0	10299
2014-03	28903	0	5338

Figure 4. The Growth History cursor.

Building a Location String

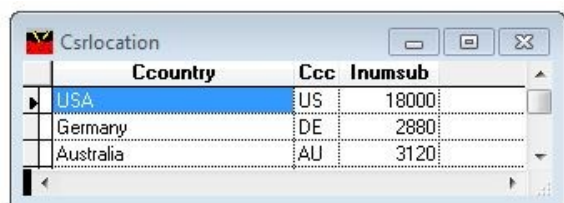
Just as interesting to us numbers junkies is the geographical distribution of a list. The 'location' method returns a tuple much like the growth history:

```
{{"country": "USA", "cc": "US", "percent": 75.0, "total": 18000},
 {"country": "Germany", "cc": "DE", "percent": 12.0, "total": 2880},
 {"country": "Australia", "cc": "AU", "percent": 13.0, "total": 3120}}
```

So, a similar parsing method would give you a cursor of countries and their relative counts:

```
if lcAction = 'loc'
  liHowManyCountries ;
  = occurs('country', luResult)
  create cursor csrLocation ;
  (cCountry c(20), cCC c(2), iNumSub i)
  y = luResult
  for li = 1 to liHowManyCountries
    lcCountry ;
    = substr(y, at('country', y, li)+10, ;
      (at('cc', y, li)-3) ;
      - (at('country', y, li)+10))
    lcCC ;
    = substr(y, at('cc', y, li)+5, ;
      (at('percent', y, li)-3) ;
      - (at('cc', y, li)+5))
    liNumSub ;
    = val(substr(y, at('total', y, li)+7, ;
      (at('}', y, li)) ;
      - (at('total', y, li)+7)))
    insert into csrLocation ;
    (cCountry, cCC, iNumSub) ;
    values ;
    (lcCountry, lcCC, liNumSub)
  next
endif
```

See **Figure 5** for an example of the cursor created.



Ccountry	Ccc	Inumsub
USA	US	18000
Germany	DE	2880
Australia	AU	3120

Figure 5. The Location cursor.

So this gives you more in-depth coverage of using the MailChimp API to perform tasks via VFP instead of using their Web site. But wait, there's more! You perhaps noticed that we didn't

cover how to update a list member's profile; doing so involves using more complicated data structures. In the next article, I'll show you how to do so, as well as executing other even more complex methods, using multiple structs and parms of various formats, and how to handle errors that may be thrown. Stay tuned!

Source Code Notes

The source code for this article is found in the subscriber downloads. It consists of a single PRG that contains a DO CASE structure for each of the various methods discussed, and a variable that causes the appropriate CASE clause to be fired.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com

Whil,

So glad to hear that you are already working on an article. My current needs are simple. Just be able to send a mail list from my VFP management system and auto trigger Mail Chimp to send it. It would assume that the customer has already created the email form and the "mail merge" fields for that form.

My system totally manages auto body shops so some of the fields would be things like car make, model, repair date, work that was done etc.

If I can do that then I'm sure my customers would ask for more but I believe in starting simple and working up from there.

I currently include SMS messaging in the system. Since the provider I use has a really solid and simple PHP API I use VFP TEXT... ENDTEXT to create the details and upload them via FTP to my website then execute a PHP script on the site that does the actual send/receive and returns the data back to VFP. All using a small app I wrote based on West Wind wwipstuff. I was thinking about doing the same thing with Mail Chimp. Everything run online today can be run with PHP and VFP9's TEXT ... ENDTEXT can create anything.

Many thanks,
John

John J. Henn, President
Jhenn Systems, Inc.
Baltimore: 410-609-0750
800-580-1627