

# Integrating Visual FoxPro and MailChimp – Part 4

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.

Welcome to the fourth article in our journey of automating MailChimp from Visual FoxPro. In this issue, I'm going to show you how to update a list member's profile by using more complicated data structures than what we've used so far. I'm also going to cover other more complex methods, using multiple structs and parms of various formats. Finally, I'll discuss how to handle errors that may be thrown.

We've learned that the basic URL formation for a call to the MailChimp API looks like this:

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

where we can sub in a DC (data center), a method, an api key (specific to your account) and an id (specific to the list you're working with.)

We can add more, such as an email address, with syntax like this:

```
&email[email]=al@example.com
```

The complete VFP code would look like this:

```
* pass one email address
if llIncludeEmail
  lcURL = lcURL ;
  ' &email[email]=' ;
  + lcEmail
endif
```

With some methods, such as batch-subscribe, you can pass multiple addresses in an array-like structure. The syntax looks like this:

```
&batch[0][email][email]=al@example.com
&batch[1][email][email]=bob@example.com
```

Note that the batch index is zero-based. Given an array, laEmail, of email addresses, the VFP code will look like this:

```
* multiple emails (batch-subscribe)
if llIncludeMultEmails
  for li = 1 to alen(laEmail,1)
    lcURL = lcURL ;
    + '&batch[' + allt(str( li-1 )) + ']' ;
    + '[email][email]=' ;
    + laEmail[li]
  next
endif
```

Assembling the entire URL string and passing it to the HTTPGet method as we've done in previous articles:

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

the luResult string looks like this:

```
{"add_count":3,"adds":
[{"email":"al@example.com","euid":
"100045b890","leid":"123456941"},
{"email":"bob@example.com","euid":
"fb323342296","leid":"196662323"},
{"email":"carla@example.com","euid":
"c2341dfs8s","leid":"194454we3"}],
"update_count":0,"updates":[],
"error_count":0,"errors":[]}
```

Remember, just like manual subscriptions, this method only causes emails to be sent to the addresses. Those people have to opt in to subscribe to the list. You won't see the addresses added to the list until they've answered the email and agreed to subscribe.

## Updating Subscriber Data

Now let's get to the update member method. First, you need to know what fields you're updating. Typically, your reason for doing the update from VFP is that the subscriber contacted you instead of doing the update themselves. So the info to be updated is per their request, and thus is going to data from the signup form they originally filled out, shown again in **Figure 1**.

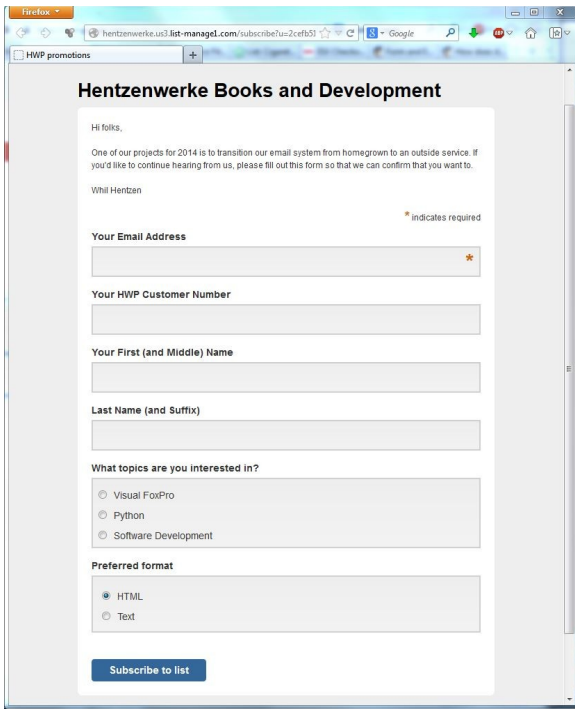


Figure 1. The original signup form.

What might a list subscriber want to have updated? Pretty much anything they filled out on the form. The format for updating a field named 'FNAME' generally looks like this:

```
lcEmail = 'al@example.com'
lcFnameNew = 'Aloyious'
lcURL = lcURL ;
+ '&email[email]=' ;
+ lcEmail + ;
+ '&merge_vars[FNAME]=' ;
+ lcFnameNew
```

### Finding Field Name Vars

The \$64,000 question that arises, of course, is "How do you find out what the 'merge vars' tag is for any other field?" The answer is "You created the name of the field when you created the sign-up form." Let's revisit. On the MailChimp site, open your list of lists (the double pieces of paper icon), then click on the name of the list of interest. You'll get the main page for the list, including the "Stats, Manage subscribers, Add subscribers, Signup forms, Settings" menu. Click on the "Signup forms" option and then the 'General forms' icon, and you'll see your signup form.

Click on a field in the form, and you'll see attributes of that form, including, in the middle, the 'field tag', which is the expression you're looking for. See Figure 2.

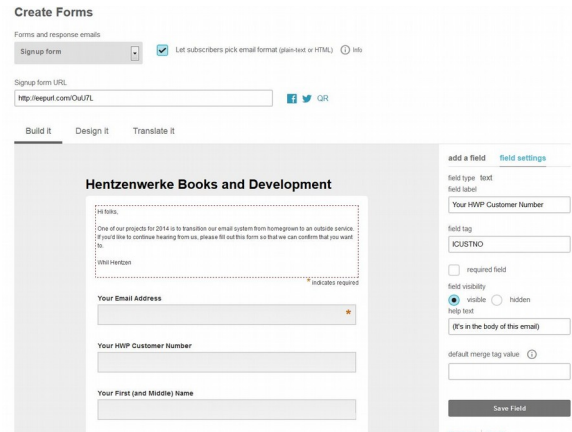


Figure 2. Editing the signup form to determine variable names.

### Individual Field Name Vars

However, I said \*generally\*. Not every field works the same. Let's look at the variations.

First, there's the email address. Every signup form has an email address, obviously, and it's always called 'EMAIL'. Technically, while you can change this field via the 'merge\_vars' clause, it's a very, very bad idea, because you would, in effect, be subscribing someone else and bypassing the double opt-in mechanism. Abuse of this mechanism is a quick way to get your MailChimp account shut down.

Next, if you've checked the "Let subscribers pick email format" checkbox to the right of the type of form (see Figure 2, near the top), a "Preferred format" option group will automatically be added to your signup form. Since it's done automatically, you don't have any control over the field. It's name is "EMAIL\_TYPE" and you can't change it.

The third type of field are free-form data entry controls like text, numbers and dates. There are a number of pre-defined data entry controls that restrict the format of what's being entered:

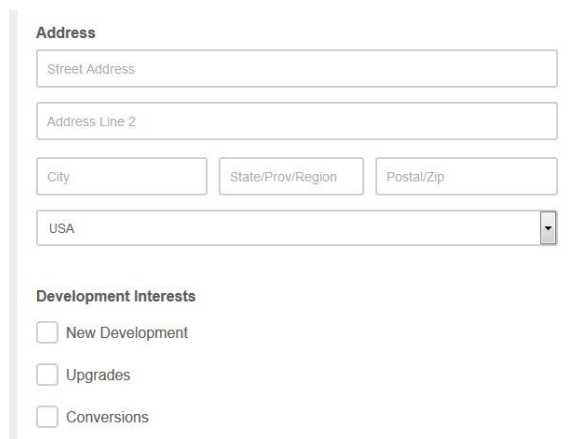
- **date** can be defined as mm/dd/ccyy or dd/mm/ccyy. There is a calendar icon on the right side of the textbox that opens a date picker.
- **birthday** can be defined as mm/dd or dd/mm, where mm is between 0 and 12 and day is between 1 and 31.
- **zip code** requires five digits.
- **phone** can be defined as international (free-form) or US/Canada (nnn-xxx-nnnn).
- **website** and **image** both require a protocol and a properly formatted URL. The image is a little awkward in that you

have to type in the URL, there isn't a mechanism to navigate to it.

All of these have a field tag that you can name during creation of the form, and thus they can all be updated using the same format as the FNAME code snippet, as long as the data being passed is valid.

The fourth type of field are multiple choice with a single allowed value - option buttons and drop downs - and you name the field, just as with free-form data entry controls. During form creation, you name the field, and the syntax is the same as the free-form data entry form. The only restriction is that the value passed must be a value you defined as an option button label or one of the items in the drop down.

The fifth type of field are checkboxes. These are a little tricky, because there is no place to define a field tag. Instead, the name of the collection of checkboxes is used as the field tag. In **Figure 3**, the name of the group of checkboxes is "Development Interests", and that's the field tag used.



The screenshot shows a form with two main sections. The first section is titled "Address" and contains five input fields: "Street Address", "Address Line 2", "City", "State/Prov/Region", and "Postal/Zip". Below these is a dropdown menu currently showing "USA". The second section is titled "Development Interests" and contains three checkboxes: "New Development", "Upgrades", and "Conversions".

**Figure 3.** Determining the field tag for complex fields.

This structure is more involved; you pass an array of values, similar to when you pass multiple email addresses to a batch-subscribe.

Finally, the granddaddy of them all, the address field. The field has a field tag, like other free-form data entry controls, that you can define. However, the address field has multiple components, as shown also in Figure 3.

You can pass a string of field values, each delimited by two spaces, to update the address. Note that every field has to be passed.

If the user doesn't enter a valid value, the field will be highlighted in red and the user will be informed how to correct. If an invalid value is entered during automatic update, however, the update is simply ignored. In some situations, the original value may be deleted. More on that later.

So that's where you find the name in the `&merge_vars[<field name>]` construct. Enough of the theory, let's look at specific examples.

### Field Name Syntax Examples

We've already seen the syntax for the FNAME field. Typically, though, we'll be updating both the first and last name at the same time.

```
lcNaFirst = 'Freddie'
lcNaLast = 'Mac'
lcURL = lcURL ;
+ '&merge_vars[FNAME]=' + lcNaFirst ;
+ '&merge_vars[LNAME]=' + lcNaLast
```

The number and zip code fields are similar. The only difference is that if the validation fails, the update will be ignored.

```
lnCustomerNumber = 123456
lcURL = lcURL ;
+ '&merge_vars[INOCUST]=' ;
+ allt(str( lnCustomerNumber ))
lcZip = '53201'
lcURL = lcURL ;
+ '&merge_vars[ZIPCODE]=' + lcZip
```

The MailChimp backend is expecting a date formatted as a string.

```
ldDate = {10/31/2014}
lcURL = lcURL ;
+ '&merge_vars[DACTIVE]=' + dtoc(ldDate)
```

A date of birth is just the month and date parts of the date, padded out to two characters.

```
ldDOB = {1/1/1980}
lcMMDD = ;
+ padl(alltrim(str(month(lddob))),2,'0') ;
+ '/' ;
+ padl(alltrim(str(day(lddob))),2,'0')
lcURL = lcURL ;
+ '&merge_vars[DOB]=' + lcMMDD
```

A phone number can either be a free-form string (for international numbers) or a strictly formatted area code, exchange and line combination. If the string passed in doesn't pass the MailChimp validation, it'll be ignored.

```
lcPhone = '414-555-1212'
lcURL = lcURL ;
+ '&merge_vars[PHONE]=' + lcPhone
```

The website and image strings need to be fully formed URLs. If they're not, the MailChimp back end will ignore them.

```
lcWebsite = 'http://www.example.com'
lcURL = lcURL ;
+ '&merge_vars[WEBSITE]=' + lcWebsite
```

A drop down field takes a string just like the free-form fields. If the string passed to the MailChimp back end isn't one of the choices

defined in the drop down, the update will be ignored.

```
lcDevHQ = 'Texas'
lcURL = lcURL ;
+ '&merge_vars[DEVHQ]=' + lcDevHQ
```

An option group field works just like the drop down. If the string passed to the MailChimp backend isn't one of the choices defined in the option group, the update will be ignored.

```
lcPrimarySoftwareInterest = 'Python'
lcURL = lcURL ;
+ '&merge_vars[PRISOFTINT]=' ;
+ lcPrimarySoftwareInterest
```

The checkbox field works like a combination of a group of email addresses as well as an option group or drop down, in that multiple values can be passed as an array, but the values are pre-defined.

Suppose you wanted to update the Development Interests field for the subscriber, specifically, to flag two of the three possible choices, Conversions and Upgrades. The format would look like so:

```
lcURL = lcURL ;
+ '&merge_vars[GROUPINGS][0][name]=' ;
+ 'Development Interests' ;
+ '&merge_vars[GROUPINGS][0][groups][0]=' ;
+ 'Conversions' ;
+ '&merge_vars[GROUPINGS][0][name]=' ;
+ 'Development Interests' ;
+ '&merge_vars[GROUPINGS][0][groups][1]=' ;
+ 'Upgrades'
```

If an invalid name for the checkbox group or an individual checkbox label is passed, the update will be ignored.

The address field consists of a string of each field, address line 1, address line 2, city, state, zip and two digit country code, concatenated and separated by two spaces.

```
lcURL = lcURL ;
+ '&merge_vars[FULLADDR]=' ;
+ '123 Main Ste 4 MyCity TX 70601 US'
```

Using four spaces between address line 1 and city (when an update doesn't have a second address line) will validate correctly.

```
lcURL = lcURL ;
+ '&merge_vars[FULLADDR]=' ;
+ '4000 Elm Ave NoCity WI 54321 US'
```

Omitting any of the fields or not delimiting all fields by two spaces will result in the entire update being ignored.

The address field collection has a second construct that you may find to be easier to use.

```
lcURL = lcURL ;
```

```
+ '&merge_vars[FULLADDR][addr1]=Corner of ' ;
+ '&merge_vars[FULLADDR][addr2]=NoAndWhere' ;
+ '&merge_vars[FULLADDR][city]=Nowhere' ;
+ '&merge_vars[FULLADDR][state]=AK' ;
+ '&merge_vars[FULLADDR][zip]=53201' ;
+ '&merge_vars[FULLADDR][country]=US'
```

There are a pair of tricks to using this construct successfully. The names of the fields, such as 'addr1', have to be submitted in lower case, and all fields have to be included in the construct, even if they're blank.

## Error Handling

So what happens if errors occur? Let's first talk about the types of errors that you may run into, and then how to deal with them.

### Types of Generated Errors

You will typically run into three types of errors.

The first is when the MailChimp server doesn't respond. These are easy to spot and (generally) easy to deal with.

The second is when a badly formatted string is sent to the server and an invalid request is sent to the server. This generates an error code and a return value that can be parsed and dealt with.

The third is when bad data is sent to the server. This also generates an error code and a return value that can be parsed and dealt with.

There is a fourth scenario that can occur, and it's difficult to deal with. In some situations, invalid data sent to the server is ignored, and doesn't generate either an error code or a return value. These are tough to deal with. Let's look at examples of each.

### Server Connection Problems

Suppose you send bad parameters in the string that identifies the server. As discussed earlier, this string looks like this

```
lcDC = 'us3'
lcMethod = 'subscribe'
lcAPIkey = 'very secret'
lcListID = 'also_very_secret'
lcURL ;
= "https://" ;
+ lcDC ;
+ ".api.mailchimp.com/2.0/lists/" ;
+ lcMethod ;
+ "?apikey=" ;
+ lcAPIkey + "-" + lcDC ;
+ "&id=" + lcListID
```

Suppose that an invalid data center value was assigned, like so:

```
lcDC = 'BadDC'
```

Since this string is part of the domain name, the server itself can't be identified, and a

connection can't be made. Web Connection's o.HTTPGet method doesn't make a connection, and the o.error property is assigned the value

**A connection with the server could not be established**

This will take a bit of manual fiddling but once you get the string fixed, you shouldn't have any troubles.

A variation of bad parameters passed would be when a bad folder name is passed. Instead of

```
+ ".api.mailchimp.com/2.0/lists/"
```

suppose a typo passed this:

```
+ ".api.mailchimp.com/1.0/lists/"
```

which isn't valid. The connection with the server is successful, and the MailChimp error handling mechanism populates the o.error property of

**Not Found**

and passes a return value of

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /1.0/lists/update-member
was not found on this server.</p>
</body></html>
```

## Internal Server Errors

There are a whole suite of situations that can generate an "Internal Server Error" error. Fortunately, the luResult return value is chock full of details about what went wrong.

Suppose you sent a bad lcListID value. The return value would look like this:

```
{"status": "error", "code": 200, "name": "List_Does
NotExist", "error": "Invalid MailChimp List ID:
24dXXMyBadListID9fa90e8"}
```

Another example would be forgetting to send an email address to a call that needs one, such as unsubscribe. The result is

```
{"status": "error", "code": -
100, "name": "ValidationError", "error": "You must
specify a email value"}
```

A third example would be sending a non-existent address to a call that needs one, such as update-member:

```
{"status": "error", "code": 232, "name": "Email_Not
Exists", "error": "There is no record of the
email address \"herman@softwaremuscle.com\" in
your account"}
```

## Errors Resulting from Bad Data

There are other errors that generate an Internal Server Error, but are caused by passing invalid data strings. For example, strings that have been assigned in the signup form, such as the labels in option groups and checkboxes, need to be matched exactly. If not, an error is passed. Suppose the name of a group of checkboxes is "Development Interests" but the string passed is missing the space between the two words.

Instead of this:

```
+ '&merge_vars[GROUPINGS][0]
[name]=DevelopmentInterests' ;
+ '&merge_vars[GROUPINGS][0][groups]
[0]=Conversions' ;
```

this is passed

```
+ '&merge_vars[GROUPINGS][0][name]=Development
Interests' ;
+ '&merge_vars[GROUPINGS][0][groups]
[0]=Conversions' ;
```

The return value looks like this:

```
{"status": "error", "code": 270, "name": "List_Inva
lidInterestGroup", "error": "\"DevelopmentIntere
sts\" is not a valid Interest Grouping name
for the list: Test List"}
```

Similarly, if "Upgrading" is errantly passed:

```
+ '&merge_vars[GROUPINGS][0][name]=Development
Interests' ;
+ '&merge_vars[GROUPINGS][0][groups]
[0]=Upgrading'
```

instead of the correct value of "Upgrades":

```
+ '&merge_vars[GROUPINGS][0][name]=Development
Interests' ;
+ '&merge_vars[GROUPINGS][0][groups]
[0]=Upgrades'
```

the luResult returned is

```
{"status": "error", "code": 270, "name": "List_Inva
lidInterestGroup", "error": "\"Upgrading\" is
not a valid Interest Group in
Grouping \"Development Interests\" on the
list: Test List"}
```

These are all easily resolved, as the return value identifies the value in question. There are issues that are more complicated. For example, supposed a badly formatted email address is passed, like so:

```
&email=[{"cemail": "muscle@softwaremuscle.com"}
]
```

instead of the correct string

```
&email[email]=muscle@softwaremuscle.com
```

and the value returned is:

```

{"status":"error","code":-
100,"name":"ValidationError","error":"Validati
on error: {\\"merge_vars\\":\\"Please enter a
struct\\\/associative array\\"}"}

```

## Bad Data Mishandled

There are some types of bad data that are passed that ignored by the MailChimp server. For example, if a bad field tag for a text box is included in a string passed to the server, the request is simply ignored. Here, the field tag "FIRSTNAME" was errantly passed instead of the correct "FNAME" tag:

```

lcURL = lcURL ;
+ '&merge_vars[FNAME]=Bernie&[LNAME]=Macoid'

```

instead of

```

lcURL = lcURL +
'&merge_vars[FNAME]=Bernie&merge_vars[LNAME]=M
acoid'

```

This request silently fails. It does not, but doesn't return an error either.

In other situations, the request does have an effect, but not the one you'd like.

For example, an address string must have each field separated by two spaces. If a single space is used between two fields, or a field is left out, the request causes the field contents to be deleted, but no error is generated.

These are both invalid strings

```

* only one space between state, zip and
country
123 MailChimp St Suite 1 Milwaukee WI 53201
US
* missing country code
123 MailChimp St Suite 1 Milwaukee WI
53201

```

and they both cause the existing data to be deleted.

On the other hand, passing certain kinds of invalid data, such as a bad state abbreviation, is ignored. This address string

```

* WX is not a valid state abbreviation
123 MailChimp St Suite 1 Milwaukee WX
53201 US

```

is allowed and is stored to the list.

## Dealing With Errors

As you can see, it's best to examine the o.Error property for a value, and if it's populated, look at luResult value for specific messages. This isn't a cure-all, but it'll take care of the more heinous problems.

A structure like this first looks for an error message:

```

lcErrorMsg = o.cErrorMsg
if !empty(lcErrorMsg)
do case
case upper(allt(lcErrorMsg)) ;
= upper('A connection with the server ' ;
+ ' could not be established')
* domain name has a problem
case upper(allt(lcErrorMsg)) ;
= upper('Not Found')
* error in the folder structure following
* the name of the domain
case upper(allt(lcErrorMsg)) ;
= upper('Internal Server Errors')
* errors in the values sent to the server
do case
case upper('Invalid MailChimp List ID') ;
$ luReturn
case upper('You must specify a email' ;
+ ' value') $ luReturn
case upper('There is no record of the ' ;
+ ' email address') $ luReturn
case upper('is not a valid Interest ' + ;
'Grouping name for the list') $ luReturn
case upper('is not a valid Interest ' + ;
' Group in Grouping') $ luReturn ;
and ;
upper('on the list') $ luReturn
case upper('Validation error: {\') ;
$ luReturn ;
and ;
upper(':\\"Please enter a struct' + ;
' \\\/associative array\') $ luReturn
endcase
otherwise
*
endcase
return
endif

```

and then does an additional search for specific strings in the luResult string if the error message was 'Internal Server Error'.

What you choose to make happen when these errors are encountered, of course, is up to you.

## Source Code Notes

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

It then provides a second CASE construct to handles the return value.

To try it out, change the variables at the top of the PRG, controlling the server parms and which action you want to perform.

## 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](mailto:whil@whilhentzen.com)*