Mailout Interactive Inc.

MyMailout / Industry Mailout Web Service Developer Documentation

Version 0.5

March 22, 2005

webservices@mymailout.com

Table of Contents

Getting Started

Introduction

The MyMailout Web Service API allows you to access your account via a simple, secure programmatic interface.

To keep up-to-date on the latest developments, please subscribe to the Developer Newsletter at http://www.industrymailout.net/Industry/Subscribe.aspx?m=1494.

Supported Operations

These include:

- Logging in
- Subscriber management: Adding, Updating, Deleting records
- Mailout management: Creating and sending email

Quick Start

To get started using the MyMailout WebServices, you need to do the following steps:

- Step 1: Obtain a MyMailout Developer account
- Step 2: Locate and import the Web Services that you want to use
- Step 3: Try the Sample Code

Step 1: Obtain a MyMailout Developer account

Please contact us for a developer account.

Step 2: Locate and import the Web Services that you want to use

Visual Studio 2003

To link to the web services from Visual Studio 2003,

1. Right click on your project
2. Select "Add a Web Reference…"
3. Enter the URL of the web service you want to connect to (see Web Service API) and select a "Web reference name" which is the namespace that you will use to refer to this service

Step 3: Try the Sample Code

There is some sample code available in the Appendices for C# and PHP.

Basic Concepts

The MyMailout System

The three most important objects in the MyMailout system are the **MailingList**, the **Mailout** and the **Subscriber**. A mailing list is simply a collection of subscribers. A mailout is an email newsletter that you will send to your list of subscribers. A subscriber is a recipient of your newsletter. Each subscriber is a member of one (and only one) list, and mailouts are similarly associated with one mailing list.

Connecting to the Web Services

There are two separate Web Services; one is for managing your list of subscribers, and one is for creating and sending Mailouts. Each web service requires separate permissions (please contact us to set these up).

Security

If you are connecting to the service via WSE2, you do not need to connect to the SSL (i.e. https) version of the web services. However, if you are using a session id, you should make sure that you are using the SSL version.

Connecting to the Subscription Management Web Service

The URL for the Subscription Management service is:

`http://www.mymailout.net/MyMailout/WebService/SubscriberManager.asmx`

The list of subscribers can be managed with the subscriber management web service. There is one subscriber list per Mailing List, and you can either update or add one subscriber at a time (for example, with a subscription form on your site) or via a batch-upload. If you have more than 100, you should use the batch-upload system.

For more information, please see the Subscriber Management API documentation

Connecting to the Mailout Management Web Service

The URL for the Mailout Management service is:

http://www.mymailout.net/MyMailout/WebService/MailoutManager.asmx

You can create and send mailouts using the Mailout Management API.

Logging In

There are two ways to provide authentication with MyMailout WebServices. If you are developing in Microsoft's .Net 2003 environment, you can use the Web Services Enhancements (WSE) 2.0, SP3. The second method is to create a new Session by calling the `Login` Web Method, then passing your Session ID as a SOAP header in subsequent Web Method calls.

Using WSE2.0

You can get the Microsoft Web Service Enhancements from their web site:

http://msdn.microsoft.com/webservices/building/wse/

Once you've got this installed, you do not need to call `Login` in order to use the web service proxy. Instead you'll need to configure the proxy class before you use it.

The process for configuring the proxy is:

1. Create a `Microsoft.Web.Services2.Security.UserToken` from your login information.
2. Create a `DerivedKeyToken` from the `UserToken`. (A `DerivedKeyToken` allows you to use a different token for multi-message exchanges, which is more secure than using the same token repeatedly.)
3. Add a `MessageSignature` element to ensure that your message is not tampered with during submission.
4. Add an `EncryptedData` element.

Here is an example of a method which configures a proxy using C#:

```
using Microsoft.Web.Services2.Security.Tokens;

using Microsoft.Web.Services2.Security;

using Microsoft.Web.Services2;
```

```
public void ConfigureProxy(WebServicesClientProtocol proxy, UsernameToken
       token)

{

       proxy.RequestSoapContext.Security.Tokens.Add(token);

              DerivedKeyToken dk1 = new DerivedKeyToken(token);



       proxy.RequestSoapContext.Security.Tokens.Add(dk1);



       // Sign the Message

       proxy.RequestSoapContext.Security.Elements.Add(

              new MessageSignature(dk1));



       // Encrypt the Message

       proxy.RequestSoapContext.Security.Elements.Add(

              new EncryptedData(dk1));

}
```

Once you have this method in place, you can use it like this:

```
// create a username token

UsernameToken usernametoken=

new UsernameToken(username, password, PasswordOption.SendHashed);



// create the proxy class to call the
```

```
// web service

SubscriberManagerWse subscriberService= new SubscriberManagerWse();



// configure the proxy with the various tokens and security

// elements

       ConfigureProxy( subscriberService, usernametoken );



// Now the WebMethods on the proxy can be called:

MailingList[] mailinglists= subscriberService.GetMailingLists();
```

Using a Session ID

If you don't require as much security as what is provided with WSE2, you can ask for a session id for use with other Web Service calls. The session id is assigned during the call to Login, and must be passed in the SOAP headers for each subsequent call. (Although it is not currently required, for future compatibility you should set the `ServerUrl` in the `SessionHeader` as well; see the example below.)

In C#, you can log in like this:

```
// create the web service proxy class:

       SubscriberManager subscriberService=new SubscriberManager();



// log in explicitly

       LoginResult loginresult= subscriberService.Login("myemail@test.zz",
       "mypassword");

       if (!loginresult.Success) {

              throw new Exception("Unable to log in");
```

```
        }


// create a SOAP header with the session id

subscriberService.SessionHeaderValue=new SessionHeader();


        subscriberService.SessionHeaderValue.SessionID=loginresult.Session
ID;
        subscriberService.SessionHeaderValue.ServerUrl=loginresult.ServerU
rl;

// now that the login is complete, the web method is called:

MailingList[] mailinglists=subscriberService.GetMailingLists();
```

Web Service API

Subscriber Management

The Subscriber Management Web Service allows you to add new subscribers to a mailing list, and update or remove existing ones. Please refer to the Connecting to the Subscriber Management Web Service section if you have already not set up the web service for use with your code.

`LoginResult Login(String username, String password)`

If you are not using WSE2, you can log in with this method.

`MailingList[] GetMailingLists()`

Retrieves the current mailing lists.

`UpdateResult SignupNewSubscriber(int mailinglistid, Subscriber subscriber, bool checkrequired)`

Add a new inactive subscriber. UpdateResult will contain a "Success" flag, or an array of error messages. You will need to send an email to confirm the subscriber's information.

`UpdateResult ConfirmSubscriber(int mailinglistid, String email)`

Confirm the subscriber information. You should call this method once the subscriber has received a confirmation email and confirmed his email address.

UpdateResult Unsubscribe(int mailinglistid, String email)

De-activate an existing subscriber. Note that this user cannot be re-activated with the Web Service interface.

int SubmitCsvFile(int mailinglistid, CsvFileDescriptor descriptor, String csvcontents)

Uploads a CSV file, but doesn't process it. See CSV Subscriber Upload Format for more information on the file format. You must provide a descriptor, which describes the columns in use in the CSV file.

void AppendCsvFile(int csvuploadid, String csvcontents)

This is not yet implemented.

void CleanupAfterCsvUpload(int csvuploadid)

Call this to clean up your temporary files and database records after completing (whether it was successful or not).

Mailout Management

The Mailout Management Web Service allows you to create and send a mailout to your MailingList. Please refer to the Connecting to the Mailout Management Web Service section if you have already not set up the web service for use with your code.

Each call may throw a SoapException if you are not logged in, or do not have permission to access a resource.

LoginResult Login(String username, String password)

If you are not using WSE2, you can log in with this method.

int CreateHTMLMailout(int mailinglistid, String subject, String html, String text)

Creates a new "Do-it-Yourself" HTML Mailout.

DeleteUnsentMailout(int mailoutid)

Delete a Mailout. You can only delete a mailout if it has not been sent yet.

`Mailout[] GetUnsentMailouts(int mailinglistid)`

Retrieve the mailouts which have not yet been sent.

`int Send(int mailoutid)`

Send the mailout to the active subscriber list.

`UploadReport ProcessCsvUpload(int csvUploadId, int maxErrors)`

Process the uploaded CSV file. This merges the upload csv file with the existing subscriber list. Erroneous and duplicate records are rejected, new records are added, and existing records are updated.

Note: the CSV upload process does not check that "required"columns are present, apart from the email address.

Objects

MailingList

The MailingList object represents a collection of subscribers. The web service only exposes the Id and the ListName.

Mailout

On the abstract level, a mailout is one email newsletter that can be sent to a list of subscribers. A mailout is created for use with one mailing list, which is specified when it is initially created.

A mailout consists of a template and some content. The relationship between the template and the content is very flexible, but usually the content is an abstract representation or "Model" of your data, and the template is the visual "View" of the content. The most common mailout setup is to add content to your mailout in the form of articles, which will be formatted together into an HTML mailout.

However, if you are creating a "Do-it-Yourself" mailout, the content is an HTML file (plus a text file), so the Template does little or no formatting. All the formatting is therefore included in your content.

A mailout may have other web pages associated with it. For example, an article may have landing pages, and a survey may have an online component, each associated with a different template.

A mailout has an Id, and a Subject.

Mailouts with distinct Template and Content

Normally, content will consist of objects such as articles, surveys, sidebar elements, persistent data, colors, fonts, styles, and so on. These are then formatted into a final HTML mailout with a template. The division between the Template and Content means that there can be two different roles for the creation of a mailout. A web designer can design a template, and then another person who may not be HTML-savvy can provide the raw content separately.

The Mailout Management Web Service does not yet provide an interface to this type of mailout.

Do-it-Yourself Mailouts

There are some features which are automatically added to your Do-it-Yourself HTML mailouts.

All DIY emails will have:

- a *forwardmessage* tag inserted after the opening body tag. (If your email is forwarded, this will appear in the forwarded email with a comment, and an indication of who forwarded the email.
- a *webbug*, which is an invisible image used for tracking open rates. This appears just before the end closing tags.

Emails created on all sites except Industry Mailout will have a footer automatically inserted with an unsubscribe, forward link, and some text indicating the "provider" site.

Emails created on Industry Mailout must include an unsubscribe link. Please see the Template section for examples of some available custom tags.

Note: Do-it-Yourself emails do indeed have templates; however, the templates do (almost) nothing but pass through the html and text content.

Subscriber

Each subscriber is a member of one list. If a person is to be a member of more than one list, he will have multiple, independent subscriber records associated with his email address.

A subscriber record is either "active" or "inactive". Once a subscriber has been de-activated, he can normally only be re-activated using the subscription form. This is a precaution to prevent people from being inadvertently added to a list from which they have opted out.

By default, the only required field is the email address, although this can be changed in the mailing list configuration via the web interface.

Template

Each mailout has one main template, and may have other templates associated with various content. The template is responsible for formatting the raw content in a "Model/View" design. (However, in the "Do-It-Yourself" HTML mailout, the template doesn't do any formatting, since layout is provided by the HTML and Text content. The template just passes the content through unmodified in this case).

There is a separate language for creating the various special features of a template, and this is fully documented in *Email Templates on MyMailout 2.0.* Here are the most common tags that you can use in your HTML mailout:

```
<mymailout:forward>Forward to a friend</mymailout:forward>

<mymailout:subscribe>Subscribe</mymailout:subscribe>

<mymailout:unsubscribe>Unsubscribe</mymailout:subscribe>

<mymailout:view>View this email online</mymailout:view>
```

You can add the attribute '`medium="text"`' to any of these tags to print out the link only. Make sure that the tags are well-formed XML and are properly closed.

Note for Advanced Users

There are two stages of parsing; the first inserts most of the content into the template, and there is a full templating language available for this. The second stage is an XSLT-like transformation, and this is when the subscriber-specific information such as link tracking URLs is inserted. Tags prefixed by `mymailout:` are second stage tags.

SOAP API

This section is not yet available.

CSV Subscriber Upload Format

This section is not yet available.

Appendix A – *A PHP Example*

A Subscription Form

- Install a PHP SOAP extension (consult the PHP documentation). This example uses nusoap and CURL for SSL. There are some calls to a form manipulation library which are not documented here. This does not include the email confirmation process.

```
<?php

	require_once( './nusoap.php');

	require_once( './forms.php');


	$action=null;



	if (isset($_POST["action"])) {

		$action=$_POST["action"];

	}

	//echo "ACTION IS $action";


	$errors=array();


	if ($action=="subscribe")

	{

		// Initialize SSL
```

```
	$ch = curl_init();

	// Log in using the list owner's id and password
	$username="myemail@test.zz";
	$password="mypassword";
	$mailinglistid=//put your mailing list id here;

	$wsdl =
"https://www.mymailout.net/Advisor/WebService/SubscriberManager.asmx?WSDL
";
	$namespace =
"http://www.mymailout.net/WebService/SubscriberManager";
	$client = new soapclient($wsdl, true);

	$params = array( "username" => $username,
				"password" => $password);

	$soap_proxy = $client->getProxy();

	//$soap_proxy->setHTTPProxy("localhost", 8080);
	$result = $soap_proxy->Login($params);
	//$result = $soap_proxy->Login($username,$password);

	$sessionid=null;
	$serverurl=null;
```

```
	if ($soap_proxy->getError())

	{

		//print ("RESULT");

		//print_r($soap_proxy->getError());

		$errors[]=$soap_proxy->faultstring;

	    //echo 'Error: ' . $soap_proxy->getError() . "\n";

	}

	else

	{

		$sessionid=$result['LoginResult']['SessionID'];

		$serverurl=$result['LoginResult']['ServerUrl'];



		//echo "THE SESSIONID IS $sessionid<br>";

		$soap_proxy->setHeaders("<SessionHeader
xmlns=\"$namespace\"><SessionID>$sessionid</SessionID><ServerUrl>".htmlsp
ecialchars($serverurl)."</ServerUrl></SessionHeader>");

		//$soap_proxy->setHeaders("<SessionHeader
xmlns=\"$namespace\"><SessionID>$sessionid</SessionID></SessionHeader>");



		$subscriber = array( "Email" => mmo_unquote("email"),

				"GivenName" =>
mmo_unquote("firstname"),

				"FamilyName" =>
mmo_unquote("lastname"),

				"IPAddress" =>
$_SERVER['REMOTE_ADDR']);
```

```
		$params=array("mailinglistid" => 1,
				"subscriber" => $subscriber,
				"checkrequired" => true);
		$updateresult=$soap_proxy->SignupNewSubscriber($params);

		//print($soap_proxy->faultstring);

		if ($soap_proxy->getError())
		{
			$errors[]=$soap_proxy->faultstring;

			//print();
		}
		else
		{
			//print_r($updateresult);
			//print ("SUCCESS: " .
$updateresult["SignupNewSubscriberResult"]["Success"] . "<br>");
			// nusoap seems to interpret boolean word
"true/false" as as string
			$success=false;
			if
($updateresult["SignupNewSubscriberResult"]["Success"]=="true") {
				$success=true;
			}
```

```
                    if ($success)

                    {

                           //echo "SUCCESS";



                    } else {

                           //echo "ERROR";

                           // nusoap is interpreting an array with one
element as a string.

                           // This fixes that.

                           //print_r($updateresult);


      $subscribeerrors=$updateresult["SignupNewSubscriberResult"]["Error
Messages"]["string"];

                           if (!is_array($subscribeerrors))

                           {


      $subscribeerrors=array($subscribeerrors);

                           } else {

                                  $subscribeerrors=$subscribeerrors;

                           }


      //print_r($updateresult["SignupNewSubscriberResult"]["ErrorMessage
s"]["string"]);

                           $errors=array_merge($errors,
$subscribeerrors);



                    }
```

```
                    }

                }

        }



?>

<html>

<head>

        <title>Subscription Demo</title>

</head>

<body>

<?php if (isset($errors) && count($errors) > 0) { ?>

<table align="center" width="400">

  <tr>

     <td style="color:#f00">The following errors were reported:</td>

  </tr>

  <?php foreach ($errors as $error) { ?>

        <tr><td style="color:#f00">&bull; <?php print $error; ?></td></tr>

  <?php } ?>



</table>

<?php } ?>

<form name="subscriptiondemo" method="post" action="<?php echo
        $_SERVER['PHP_SELF']; ?>">

<input type="hidden" name="action" value="subscribe" />
```

```
<table align="center" style="border:none; border:2px dotted black;">

  <tr>

    <td>Email (required)</td>

    <td><input name="email" value="<?php print mmo_formvar("email"); ?>"
       /></td>

  </tr>

  <tr>

       <td>First Name</td>

       <td><input name="firstname" value="<?php print mmo_formvar("firstname");
       ?>" /></td>

  <tr>

       <td>Last Name</td>

       <td><input name="lastname" value="<?php print mmo_formvar("lastname");
       ?>" /></td>

  <tr>

       <td colspan="2" align="center"><input type="submit" name="btnsubmit"
       value="Subscribe" /></td>

  </tr>



</table>

</form>

</body>

</html>
```

Appendix B – *A C# Example*

Upload Subscribers, Send a Do-it-yourself Mailout

```
using System.Text;

using System.IO;


using MyMailoutTests.SubscriberManagerService; // the subscriber manager web
       service

using MyMailoutTests.MailoutManagerService; // the mailout manager web service


using Microsoft.Web.Services2.Security.Tokens;

using Microsoft.Web.Services2.Security;

using Microsoft.Web.Services2;


namespace MyMailoutTests.WebService

{


       public class Example

      {

              public void ExampleTestWse()

              {

                     /// UPLOAD A SUBSCRIBER LIST

                     // create the Subscriber Manager proxy class
```

```
SubscriberManagerWse subscriberService=new
       SubscriberManagerWse();

//subscriberService.Proxy=new
       System.Net.WebProxy("http://127.0.0.1:8080/", false);



// create a WSE2 username token

UsernameToken usernametoken=new
       UsernameToken("test@test.zz", "mypassword",
       PasswordOption.SendHashed);



// configure the proxy (see "Using WSE2.0").

_wsesecurityhelper.ConfigureProxy(subscriberService,
       usernametoken );



// retrieve the mailing lists and use the first one. (Here,

// we just use the first one, but normally we would know in

// advance which one to upload to.)

// "SubscriberManagerService" is the Namespace we imported
       the webservice

// with.

SubscriberManagerService.MailingList[] mailinglists=null;

try

{

       mailinglists=subscriberService.GetMailingLists();

}

catch (Exception ex)

{
```

```
        System.Console.Error.Write("ERROR: "+ex.Message);

        throw ex;

}

if (mailinglists.Length<1)

{

        throw new Exception("No mailing lists are
                configured");

}

SubscriberManagerService.MailingList
        mailinglist=mailinglists[0]; // just select the first
        mailing list



// read the file from the harddrive

FileInfo fileinfo=new FileInfo(_filename);

Assert.IsTrue(fileinfo.Exists, "Can't find "+_filename);

StreamReader sr=new StreamReader(fileinfo.OpenRead());

String csvContents=sr.ReadToEnd();



// Describe the file we just loaded

CsvFileDescriptor descriptor=new CsvFileDescriptor();


descriptor.EmailColumn=1; // the leftmost column is "1",
        not "0".

descriptor.GivenNameColumn=2;

descriptor.FamilyNameColumn=3;

descriptor.TextOnlyColumn=4; // don't have a text-only
        column
```

```
        descriptor.HasHeader=true; // does the first column have a
               header?


        // Upload the file (Note: SubmitCsvFile only puts the file
               on the remote
        // server, without processing it.
        int csvUploadId=-1;
        UploadReport report=null;
        try
        {

csvUploadId=subscriberService.SubmitCsvFile(mailinglist.Id,
        descriptor, csvContents);

report=subscriberService.ProcessCsvUpload(csvUploadId, 200);
        }
        catch (Exception ex)
        {
               System.Console.Error.Write("ERROR: "+ex.Message);
               throw ex;
        }

        // process the file we just uploaded, with a maximum of 200
               error messages
        // of each type (i.e. 200 duplicate email addresses, 200
               invalid email address, etc.
```

```
System.Console.Out.Write(FormatReport(report));


// remove the intermediate records for the upload.

subscriberService.CleanupAfterCsvUpload(csvUploadId);


/// SEND THE EMAIL

String htmlcontent="<html><body><p><em>THIS IS A
       TEST</em></p><p><mymailout:unsubscribe>Unsubscribe</m
       ymailout:unsubscribe></body></html>";

String textcontent="THIS IS A TEST \r\n\r\nUnsubscribe:
       <mymailout:unsubscribe medium=\"text\"/>";

String subject="This is a test";


// create a proxy to the Mailout Manager.  This is separate
       from

// the Subscriber Manager, but we can use the same username
       token,

// and the mailing list id we used above is still the same.

MailoutManagerWse mailoutService=new MailoutManagerWse();

//mailoutService.Proxy=new
       System.Net.WebProxy("http://127.0.0.1:8080/", false);


_wsesecurityhelper.ConfigureProxy(mailoutService,
       usernametoken);


// create the mailout
```

```
        int
               mailoutid=mailoutService.CreateHTMLMailout(mailinglis
               t.Id, subject, htmlcontent, textcontent);



        // send it!

        int sent=mailoutService.Send(mailoutid);

        System.Console.Out.Write("Sent mailout: "+subject);

        System.Console.Out.Write("List name:
               "+mailinglist.ListName);

        System.Console.Out.Write("Email sent:   "+sent);



}



/// <summary>

/// Create a readable version of the upload report

/// </summary>

/// <param name="uploadreport"></param>

/// <returns></returns>

private String FormatReport(UploadReport uploadreport)

{

        StringBuilder sb=new StringBuilder();

        sb.Append("UPLOAD SUMMARY FOR
               "+uploadreport.Sourcename+"\r\n");



        sb.Append("Total Records found in file:
               "+uploadreport.Totaluploaded+"\r\n\r\n");
```

```
        sb.Append("ADDED:\r\n");

        log.Debug("Total existing records updated:
              "+uploadreport.Totalupdated+"\r\n");

        log.Debug("Total new records inserted:
              "+uploadreport.Totalinserted+"\r\n");

        sb.Append("NOT ADDED:\r\n");

        sb.Append("Total Invalid Records:
              "+uploadreport.Totalinvalid+"\r\n");

        sb.Append(FormatErrors(uploadreport.Invalidemailerrors));

        sb.Append("Total Duplicate Records Uploaded:
              "+uploadreport.Totalduplicates+"\r\n");

        sb.Append(FormatErrors(uploadreport.Duplicateemailerrors));

        sb.Append("Total Unsubscribed and not Re-added:
              "+uploadreport.Totalunsubscribed+"\r\n");

        sb.Append(uploadreport.Alreadyunsubscribederrors);

        return sb.ToString();

}



/// <summary>

/// Create a multi-line string with one error per line,

/// including the line number, email address, and error

/// message.

/// </summary>

/// <param name="errors"></param>

private String FormatErrors(UploadError[] errors)

{
```

```
            StringBuilder sb=new StringBuilder();

            foreach (UploadError error in errors)

            {

                  String email=error.Email;

                  if (email==null || email.Trim()=="")

                  {

                        email="(empty)";

                  }

                  sb.Append("   #"+error.Rownumber+" <"+email+">:
                        "+error.Errormsg+"\r\n");

            }

            return sb.ToString();

      }


}
```

Appendix C – *Errors*

You do not have sufficient permission to access this page – The account needs to be set up with Developer access.