dialogue

SMS REST API Toolkit

C# Quick Start Guide

Date: March 2013

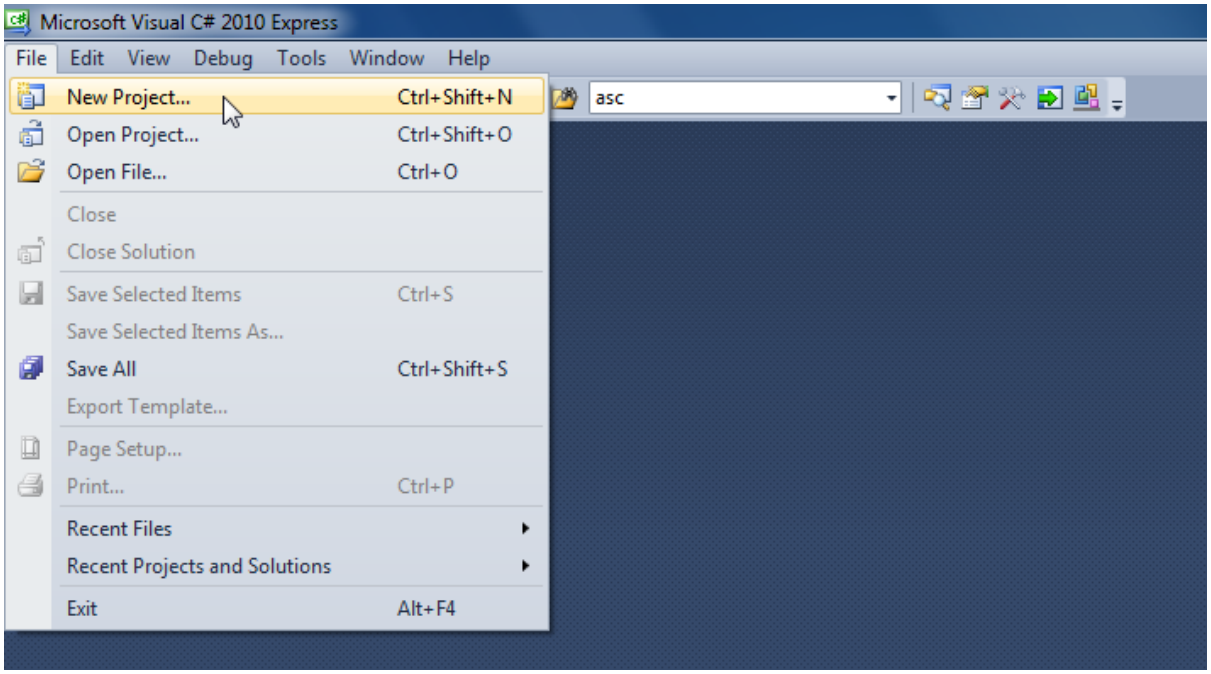
Version: 1.6.4

CONTENT

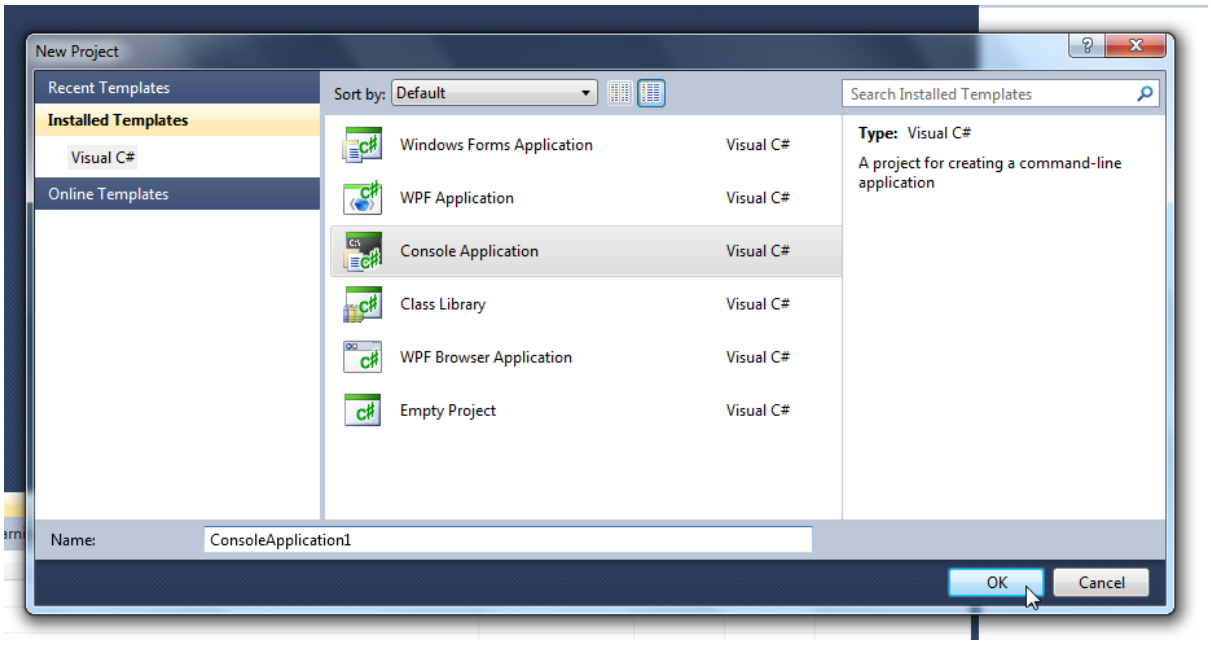
1 Sending Messages

1.1 Creating the Project

Using Microsoft Visual C# 2010 (Express) create a new project.

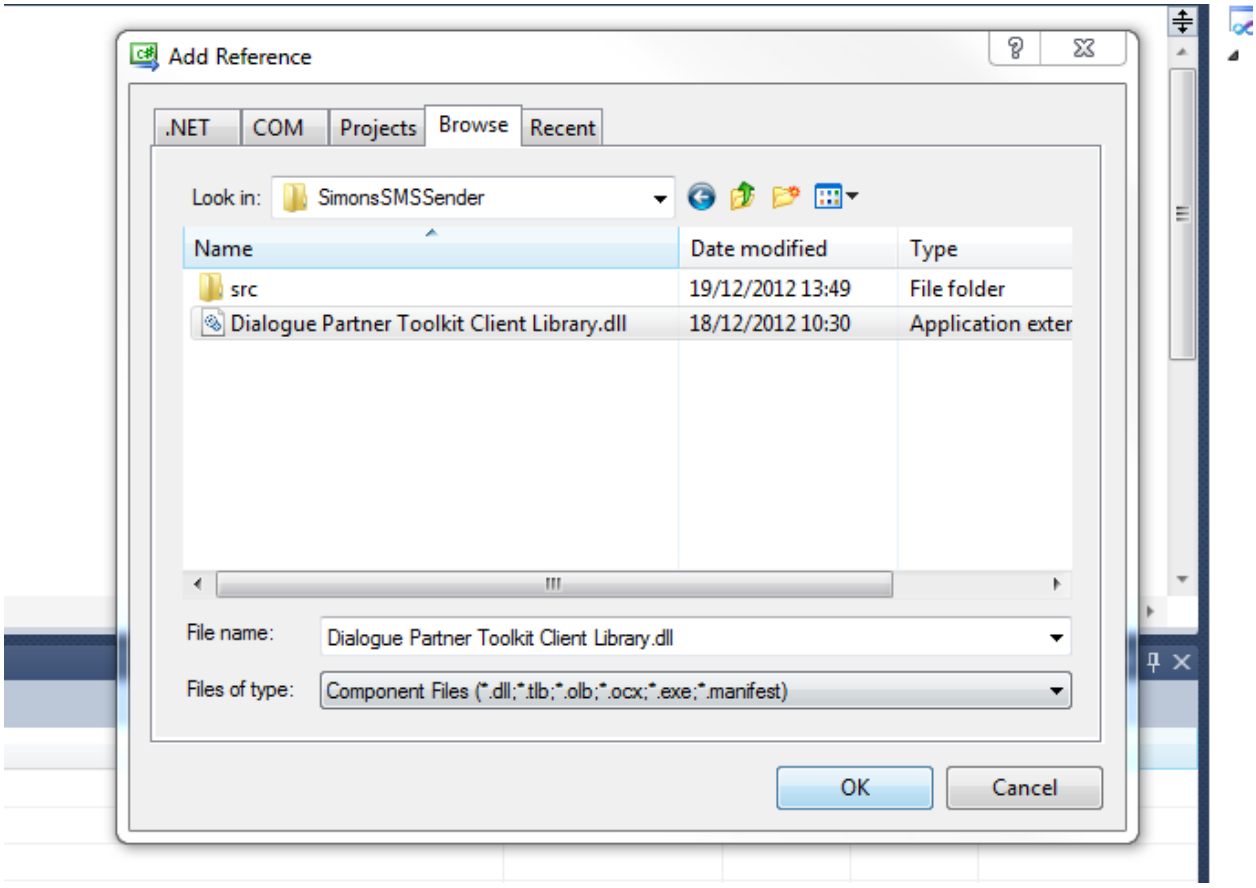


Select your desired project type, for instance **Console Application**, and give the project a name.

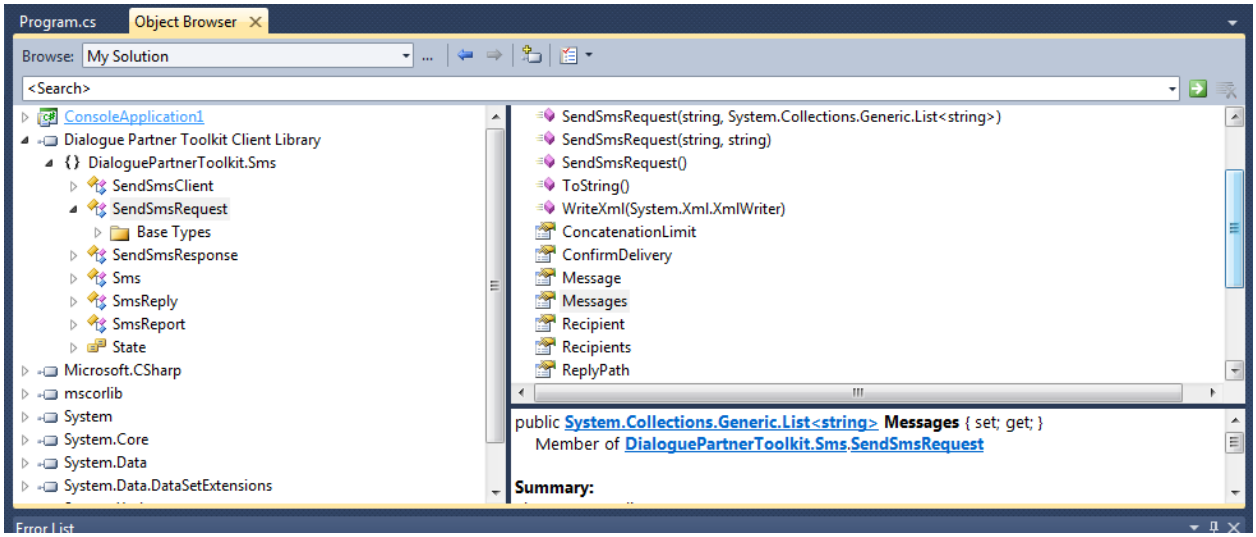


Click **OK** and wait until the project is created. When ready you will see a program template generated for you, e.g. Program.cs.

To add the toolkit library to your project open menu **Project** > **Add Reference...** and select the **Browse** tab. Navigate to the folder containing **Dialogue Partner Toolkit Client Library.dll** and select this file.



Click **OK** to add the library to your project. You can see a Dialogue Partner Toolkit Client Library entry under **References** in the **Solution Explorer** (right hand side). You can also visualize the libraries' objects and their fields using the Object Browser (right-click the library reference and select **View in Object Browser**). Take a look at the `SendSmsRequest` object.



1.2 Adding the Namespace

The namespace required for sending messages is `DialoguePartnerToolkit.Sms`. Here are the objects of interest for sending messages:

Object	Description
`SendSmsClient`	Allows sending of messages
`SendSmsRequest`	Input object (message parameters)
`SendSmsResponse`	Output object (submission results)
`Sms`	Contained once or more inside SendSmsResponse

To use this namespace include `using DialoguePartnerToolkit.Sms;` in your source.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using DialoguePartnerToolkit.Sms;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

1.3 Instantiating SendSmsClient

To send a message you must first instantiate the `SendSmsClient` object and pass it your API endpoint and API credentials (user name and password) using the `Endpoint` and `Credentials` properties:

```
static void Main(string[] args)
{
    SendSmsClient client = new SendSmsClient()
    {
        Endpoint = "endpoint",
        Credentials = new System.Net.NetworkCredential("user", "pass")
    };
}
```

Replace the yellow marked `endpoint`, `user` and `pass` by your API endpoint host name, user name and password, respectively. You will have received your API credentials as part of the Dialogue sign-up process. You will also have been given a document or link to a document, which contains the endpoint to use. Do not prefix the `Endpoint` property with http://.

By default secure communication via SSL is enabled; to disable SSL you can set the `Secure` property to false; however we strongly recommend to leave secure communication enabled.

1.4 Sending a Message

To send a message, first instantiate the `SendSmsRequest` object, which defines various message parameters. For example:

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxxx"
};
```

Replace the yellow marked `44xxxxxxxxxx` by your recipient number in international, normalized format, e.g. starting with 1 for US numbers or 44 for UK numbers.

To submit the message write:

```
SendSmsResponse response = client.SendSms(request);
```

Run your application and that's it! You've just sent an SMS message.

1.4.1 Complete Code Sample

Here is the full code again:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using DialoguePartnerToolkit.Sms;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            SendSmsClient client = new SendSmsClient()
            {
                Endpoint = "endpoint",
                Credentials = new System.Net.NetworkCredential("user", "pass")
            };

            SendSmsRequest request = new SendSmsRequest()
            {
                Message = "This is a test message",
                Recipient = "44xxxxxxxxxx"
            };
```

```
            SendSmsResponse response = client.SendSms(request);
        }
    }
}
```

1.5 Processing the Response

You should inspect the `SendSmsResponse` object returned by the `SendSms` call to see if the submission was successful or a failure. The response contains a list of `Sms` objects, each one corresponding to a submitted message. The `Recipient` property can be used to associate a submission result with the original recipient (in case multiple recipients were specified).

The `Successful` property of each `Sms` object is `true` if the submission succeeded (in which case the `Id` property contains the unique submission identifier); otherwise use properties `SubmissionReport` and `ErrorDescription` to get an error code and description, respectively.

Here is some code that inspects the `SendSmsResponse` object:

```
foreach (Sms sms in response.Messages)
{
    if (sms.Successful)
    {
        Console.WriteLine("Submission to '{0}' successful; " +
            "messageId: {1}",
                sms.Recipient,
                sms.Id
        );
    }
    else
    {
        Console.WriteLine("Submission to '{0}' failed; " +
            "errorCode: {1}, errorDescription: {2}",
                sms.Recipient,
                sms.SubmissionReport,
                sms.ErrorDescription
        );
    }
}
```

To stop the console window closing immediately after the program executes you should add `Console.ReadKey();` after the `foreach { ... }` loop, which will wait for any key being pressed to continue.

1.5.1 Complete Code Sample

Here is the full code again:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using DialoguePartnerToolkit.Sms;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            SendSmsClient client = new SendSmsClient()
            {
                Endpoint = "endpoint",
                Credentials = new System.Net.NetworkCredential("user", "pass")
            };

            SendSmsRequest request = new SendSmsRequest()
            {
                Message = "This is a test message",
                Recipient = "44xxxxxxxxxx"
            };

            SendSmsResponse response = client.SendSms(request);

            foreach (Sms sms in response.Messages)
            {
                if (sms.Successful)
                {
                    Console.WriteLine("Submission to '{0}' successful; " +
                        "messageId: {1}",
                            sms.Recipient,
                            sms.Id
                    );
                }
                else
                {
                    Console.WriteLine("Submission to '{0}' failed; " +
                        "errorCode: {1}, errorDescription: {2}",
                            sms.Recipient,
                            sms.SubmissionReport,
                            sms.ErrorDescription
                    );
                }
            }

            Console.ReadKey();
        }
    }
}
```

1.6 Multiple Messages

You can submit multiple messages at once by providing multiple strings to the `Messages` property, a property of type `List<string>`. You can also use the corresponding constructor to create the request object with a string list.

```
            SendSmsRequest request = new SendSmsRequest()
            {
                Messages = new List<string> {
                    "This is a test message", "This is another message"
                },
```

```
    Recipient = "44xxxxxxxxx"
};
```

In the response you will find one `Sms` object for each message. The foreach loop used for processing the response in section 1.5 iterates over all returned `Sms` objects and prints the submission outcome for each message.

1.7 Multiple Recipients

You can target multiple recipients at once by providing multiple strings to the `Recipients` property, a property of type `List<string>`. You can also use the corresponding constructor to create the request object with a string list.

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipients = new List<string> {
        "44xxxxxxxxx", "44xxxxxxxxx"
    }
};
```

In the response you will find one `Sms` object for each recipient. The foreach loop used for processing the response in section 1.5 iterates over all returned `Sms` objects and prints the submission outcome for each recipient.

1.8 Additional Parameters

There are other parameters that can be provided via properties of the `SendSmsRequest` object. You can call `Console.WriteLine(request);` to dump all parameters in XML representation to the console for inspection.

1.8.1 ConcatenationLimit

By setting the `ConcatenationLimit` property, you enable long message concatenation. If the length of any message exceeds the default character limit the messaging gateway will send multiple concatenated messages up to the concatenation limit, after which the text is truncated. Concatenation works by splitting and wrapping the message in packets or fragments, each fragment being prefixed by the current fragment number and the total number of fragments. This allows the phone to know when it received all fragments and how to reassemble the message even if fragments arrive out of order.

The concatenation limit refers to the maximum number of message fragments, not the number of characters, e.g. a concatenation limit of “3” means no more than 3 SMS messages will be sent, which in total can contain up to 459 GSM-compatible characters. The concatenation overhead is 7 characters, so 160 - 7 = 153 available characters per fragment x 3 = 459 total characters.

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxx",
```

```
    ConcatenationLimit = 10
};
```

In the response you will find one `Sms` object for each message segment.

1.8.2 UserTag

By setting the `UserTag` property you can tag messages. You can use this for billing purposes when sending messages on behalf of several customers.

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxxx",
    UserTag = "Customer1"
};
```

The UserTag property must not exceed 50 characters; longer values will make the submission fail.

1.8.3 ScheduleFor

Use the `ScheduleFor` property to delay sending messages until the specified date and time. The property is of type `DateTime` and must be an instance of `DateTimeKind.Utc` so that time zone conversion is possible.

This example sends the message on 1st December 2011 at 6:30 PM:

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxxx",
    ScheduleFor = new DateTime(2011, 12, 1, 18, 30, 0, DateTimeKind.Utc)
};
```

If the schedule date/time is in the past the message is sent immediately.

1.8.4 ValidityPeriod

Use the `ValidityPeriod` property to specify the maximum message delivery validity after which the message is discarded unless received. The property is of type `TimeSpan` (seconds and milliseconds are ignored). If not set the default validity period is applied.

This example sets the validity period to 1 week:

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxxx",
    ValidityPeriod = new TimeSpan(7, 0, 0, 0)
};
```

The maximum validity period is 14 days, if you specify a longer validity period 14 days will be used.

1.8.5 ConfirmDelivery, ReplyPath, UserKey

The property `ConfirmDelivery` can be set to `true` to enable tracking of message delivery. If you enable `ConfirmDelivery` you must also set the `ReplyPath` property pointing to an HTTP event handler that you implement (see section 2). Optionally, set `UserKey` to an arbitrary, custom identifier, which will be posted back to you. You can use this to associate a message submission with a delivery report.

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxx",
    ConfirmDelivery = true,
    ReplyPath = "http://www.myserver.com/mypath",
    UserKey = "myKey1234"
};
```

Note that depending on that path setup some web servers may send redirects. For example, if `mypath` is a directory the web server may respond with a 302 redirect response indicating the post should go to `mypath/`. The messaging platform does **not** follow redirects, so you need to make sure that your reply paths do not use redirection.

1.8.6 SessionId, SessionReplyPath

The property `SessionReplyPath` points to an HTTP event handler that you have implemented (see section 2). The handler is invoked if the recipient replies to the message they have received. Optionally you can specify the `SessionId` property, which will be posted back to you. You can use this to associate a message submission with a reply.

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
    Recipient = "44xxxxxxxxx",
    SessionReplyPath = "http://www.myserver.com/mypath",
    SessionId = "1234567890"
};
```

Note that depending on that path setup some web servers may send redirects. For example, if `mypath` is a directory the web server may respond with a 302 redirect response indicating the post should go to `mypath/`. The messaging platform does **not** follow redirects, so you need to make sure that your reply paths do not use redirection.

1.9 Custom Parameters

The Dialogue Messaging Gateway supports further parameters. To be able to use these parameters the `SendSmsRequest` object is based on `Dictionary<string, string>`. For example, to supply your own unique submission identifier you can write this code:

```
SendSmsRequest request = new SendSmsRequest()
{
    Message = "This is a test message",
```

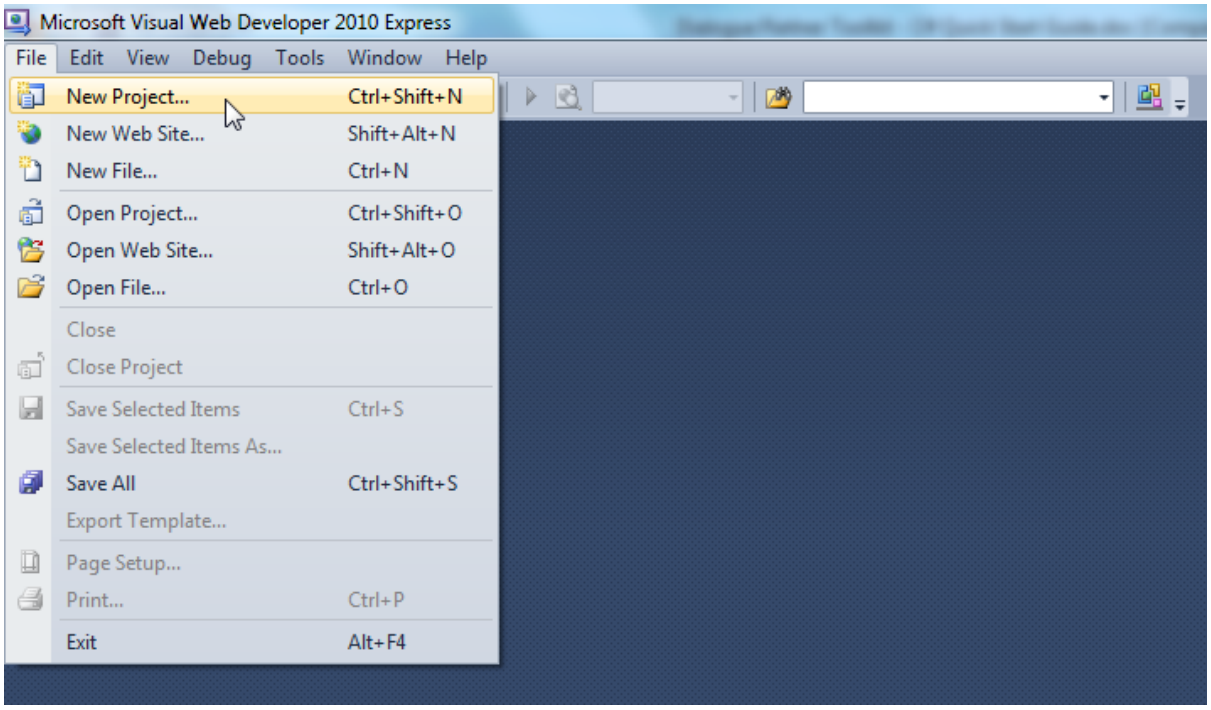
```
    Recipient = "44xxxxxxxxx"
};

request["X-E3-Submission-ID"] = Guid.NewGuid().ToString();
```

(If you use the same `X-E3-Submission-ID` parameter in a future `SendSmsRequest` object it will not perform a new message submission but return the previous `SendSmsResponse` object. Do not confuse this parameter with the returned `Sms.Id` property – they are different.)

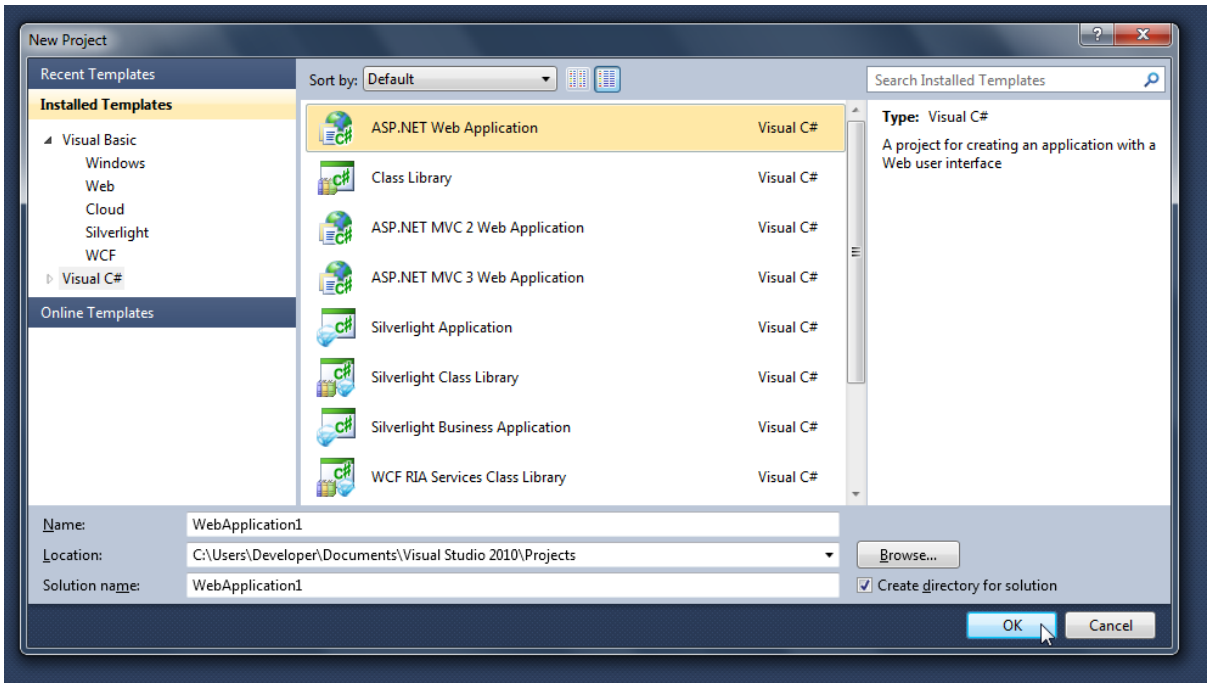
2 Receiving Reports and Replies

2.1 Creating the Project

Using Microsoft Visual Web Developer 2010 (Express) create a new project.

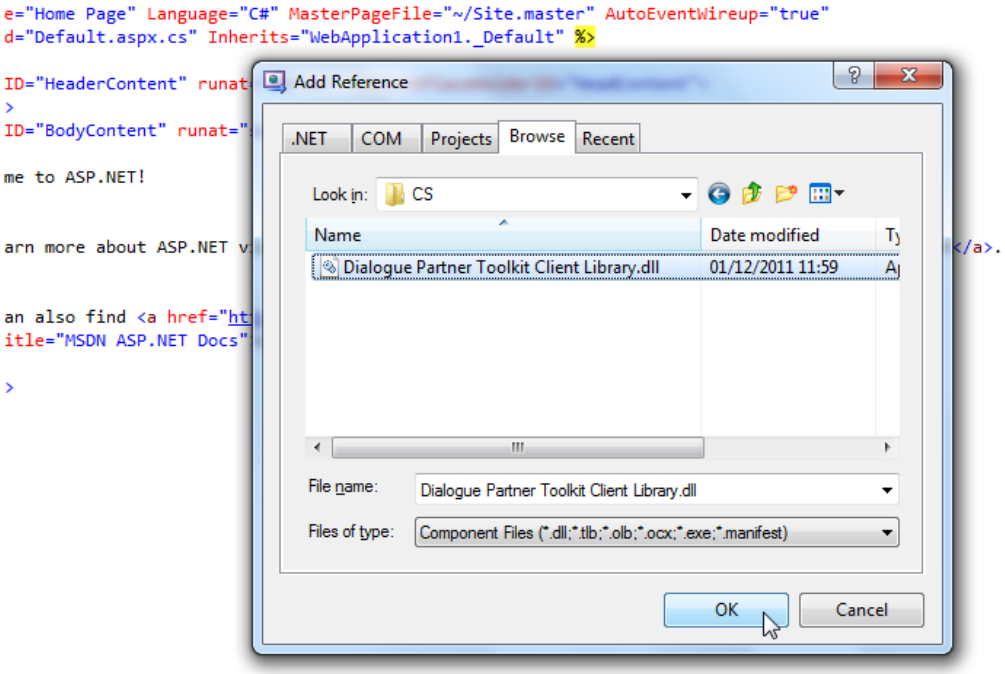


Select **Visual C#** under **Installed Templates** and **ASP.NET Web Application** | Visual C# as project type. Give the project a name.

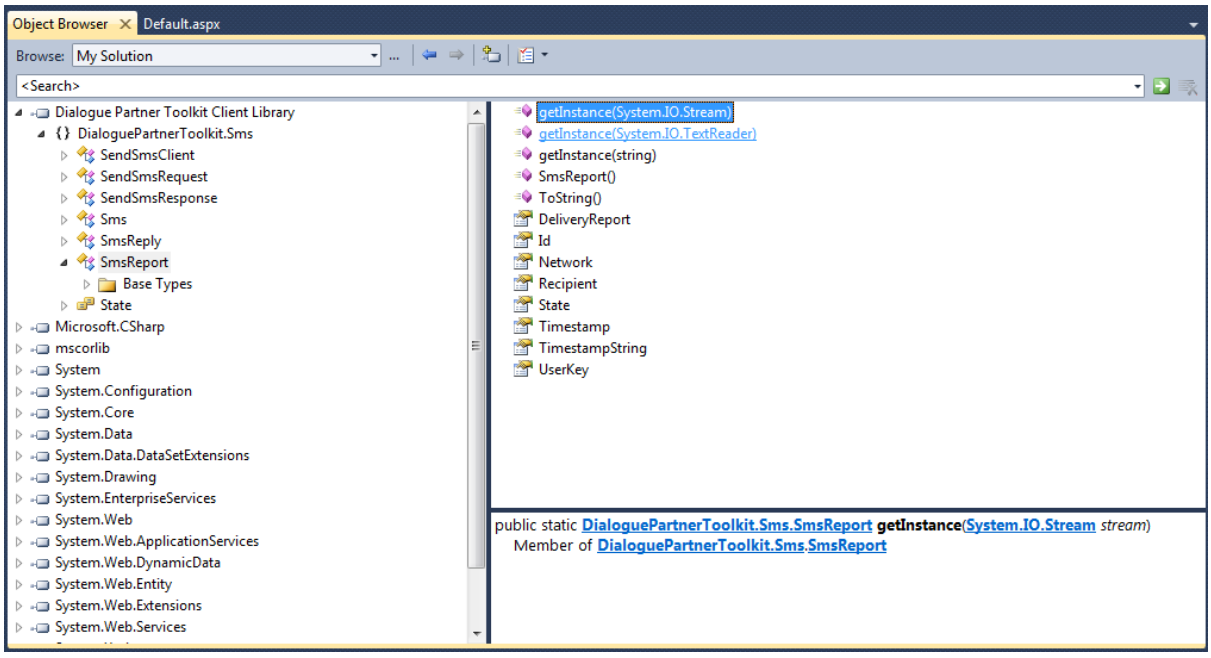


Click **OK** and wait until the project is created. When ready you will see a page template generated for you, e.g. Default.aspx.

To add the toolkit library to your project open menu **Project** > **Add Reference...** and select the **Browse** tab. Navigate to the folder containing **Dialogue Partner Toolkit Client Library.dll** and select this file.



Click **OK** to add the library to your project. You can see a Dialogue Partner Toolkit Client Library entry under **References** in the **Solution Explorer** (right hand side). You can also visualize the libraries' objects and their fields using the Object Browser (right-click the library reference and select **View in Object Browser**). Take a look at the `SmsReport` object.



2.2 Adding the Namespace

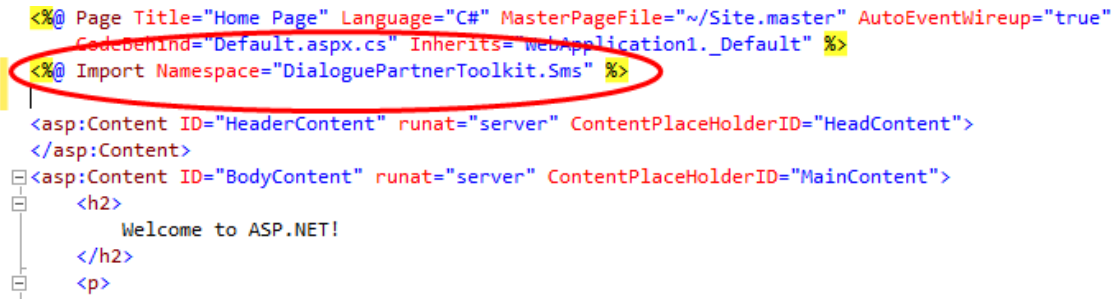
The namespace required for processing reports or replies is `DialoguePartnerToolkit.Sms`. Here are the objects of interest for receiving reports and replies:

Object	Description
`SmsReport`	Utility object for parsing incoming reports (ASP.NET)
`SmsReply`	Utility object for parsing incoming replies (ASP.NET)

To use this namespace include

```
<%@ Import Namespace="DialoguePartnerToolkit.Sms" %>
```

below the `<%@ Page... %>` directive on your page.



You can remove the `HeaderContent` section and all elements inside the `BodyContent` section; keep the `BodyContent` section wrapper itself though. (These are generated by the Visual Web Developer template but are not used.)

2.3 Processing Message Reports

To process message report POST requests on your page write:

```
<%
    if (Request.HttpMethod == "POST")
    {
        SmsReport report = SmsReport.getInstance(Request.GetBufferlessInputStream());
        switch (report.State)
        {
            case State.Delivered:
                // Message delivered
                break;
            case State.TemporaryError:
                // Temporary error but could still be delivered
                break;
            case State.PermanentError:
                // Permanent error, message was not delivered
                break;
        }
    }
%>
```

You could add some logging using:

```
System.Diagnostics.Debug.WriteLine(
    "Message delivered: " + report.Recipient + ", " + report.UserKey);
```

Run your project. Your web browser will open using an address which looks like this (the port number could vary):

http://localhost:55206/Default.aspx

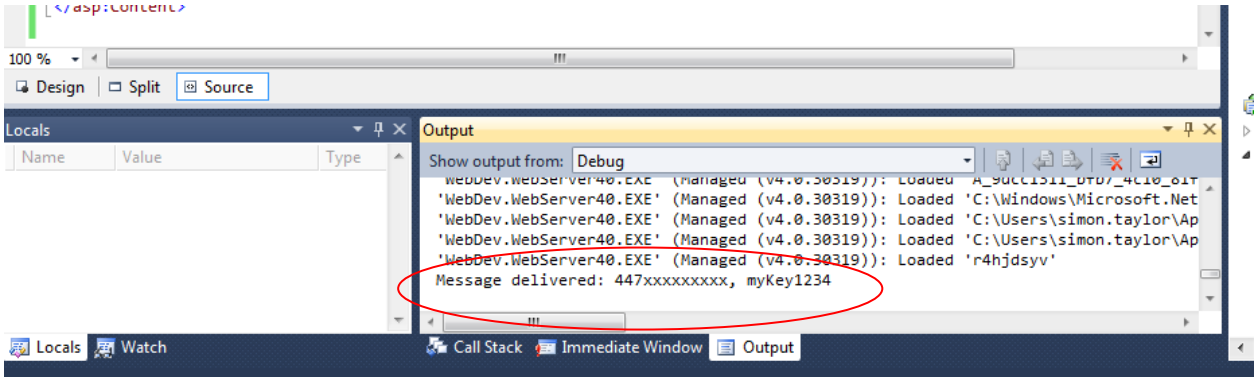
This displays a template ASP.NET page. There's no parsing of an incoming report since this is a GET request.

To test the report processing download and install the `curl` command line utility from http://curl.haxx.se. Once installed you can fake a successful delivery report by running:

```
curl -H "Content-Type: application/xml" -d "<callback X-E3-Delivery-
Report=\"00\" X-E3-ID=\"90A9893BC2B645918034F4C358A062CE\" X-E3-
Loop=\"1322229741.93646\" X-E3-Network=\"Orange\" X-E3-
Recipients=\"447xxxxxxxxx\" X-E3-Timestamp=\"2011-12-01 18:02:21\" X-E3-
User-Key=\"myKey1234\"/>" -X POST http://localhost:55206/Default.aspx
```

(Make sure the port number 55206 coincides with your own.).

The result from the curl call can be seen in the output window.



To fake a temporary error, change the X-E3-Delivery-Report value in the XML from 00 to something between 20 and 3F. To fake a permanent error, change the X-E3-Delivery-Report value to something between 40 and 7F.

That's it! You've successfully implemented a delivery report event handler.

Of course you should use the `Recipient` and/or `UserKey` properties to associate the report with a previous submission. For example, prior to submission you might store a recipient and user key record in your own database. The record would also have a DELIVERED column which is left undefined or NULL. Upon receiving a report you may look up the record using the `Recipient` and `UserKey` properties and update the DELIVERED column to 'Yes' or 'No' to reflect the delivery state.

All properties exposed by `SmsReport` are:

Property	Description
`Id`	Unique report identifier (not that of the original submission).
`Recipient`	The original recipient of the submission.
`DeliveryReport`	Delivery report value, 00 to 1F indicating a successful delivery, 20 to 3F indicating a temporary error and 40 to 7F indicating a permanent error.
`State`	The delivery report value classified into `State.Delivered`, `State.TemporaryError` or `State.PermanentError`.
`Successful`	True if `State` is `State.Delivered`, false otherwise.
`UserKey`	Optional `UserKey` parameter from the message submission. Can be used to associate unique submissions with reports, even if the recipient is the same.
`Timestamp`	Date and time the report was received (UTC).
`Network`	Mobile operator network name.

Here is the full ASP.NET page source code:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master"
AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="WebApplication1._Default" %>
<%@ Import Namespace="DialoguePartnerToolkit.Sms" %>

<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
<%
    if (Request.HttpMethod == "POST")
    {
        SmsReport report = SmsReport.getInstance(Request.GetBufferlessInputStream());
        switch (report.State)
        {
            case State.Delivered:
                // Message delivered
                System.Diagnostics.Debug.WriteLine(
                    "Message delivered: " + report.Recipient + ", " + report.UserKey);
                break;
            case State.TemporaryError:
                // Temporary error but could still be delivered
                System.Diagnostics.Debug.WriteLine(
                    "Temporary error: " + report.Recipient + ", " + report.UserKey);
                break;
            case State.PermanentError:
                // Permanent error, message was not delivered
                System.Diagnostics.Debug.WriteLine(
                    "Permanent error: " + report.Recipient + ", " + report.UserKey);
                break;
        }
    }
%>
</asp:Content>
```

Important: If you copy and paste this page source replace the reference to `WebApplication1` with your project name.

2.4 Processing Message Replies

To process reply POST requests on your page write:

<%

```
    if (Request.HttpMethod == "POST")
    {
        SmsReply reply = SmsReply.getInstance(Request.GetBufferlessInputStream());
        string sender = reply.Sender;
        string message = reply.Message;
        string sessionId = reply.SessionId;
    }
```

%>

You could add some logging by using:

```
        System.Diagnostics.Debug.WriteLine(
            sender + ", " + message + ", " + sessionId);
```

Run your project. Your web browser will open using an address which looks like this (the port number could vary):

http://localhost:55206/Default.aspx

This displays a template ASP.NET page. There's no parsing of an incoming reply since this is a GET request.

To test the reply processing download and install the `curl` command line utility from http://curl.haxx.se.

Once installed, you can fake a successful reply by running:

```
curl -H "Content-Type: application/xml" -d "<callback X-E3-Account-
Name=\"test\" X-E3-Data-Coding-Scheme=\"00\" X-E3-Hex-
Message=\"54657374204D657373616765\" X-E3-
ID=\"809EF683F022441DB9C4895AED6382CF\" X-E3-Loop=\"1322223264.20603\" X-
E3-MO-Campaign=\"\" X-E3-MO-Keyword=\"\" X-E3-Network=\"Orange\" X-E3-
Originating-Address=\"447xxxxxxxx\" X-E3-Protocol-Identifier=\"00\" X-E3-
Recipients=\"1234567890\" X-E3-Session-ID=\"1234567890\" X-E3-
Timestamp=\"2011-11-25 12:14:23.000000\" X-E3-User-Data-Header-
Indicator=\"0\"/>" -X POST http://localhost:55206/Default.aspx
```

(Make sure the port number 55206 coincides with your own.)

That's everything required to implement a handler which can receive incoming replies. Here are all the properties exposed by `SmsReply`:

Property	Description
`Id`	Unique message identifier (not that of the original submission).
`Sender`	The original recipient of the submission (now the sender as it's a reply).
`SessionId`	Optional `SessionId` parameters from submission. Can be used to associate unique submissions with replies, even if the

	recipient is the same.
Message	The message text.
Timestamp	Date and time the message was received (UTC).
Network	Mobile operator network name.

Here is the full ASP.NET page source code:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master"
AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="WebApplication1._Default" %>
<%@ Import Namespace="DialoguePartnerToolkit.Sms" %>

<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
<%
    if (Request.HttpMethod == "POST")
    {
        SmsReply reply = SmsReply.getInstance(Request.GetBufferlessInputStream());
        string sender = reply.Sender;
        string message = reply.Message;
        string sessionId = reply.SessionId;
        System.Diagnostics.Debug.WriteLine(
            sender + ", " + message + ", " + sessionId);
    }
%>
</asp:Content>
```

Important: If you copy and paste this page source replace the reference to `WebApplication1` with your project name.