

Free.iso8583 Tutorial

Written by: AT Mulyana

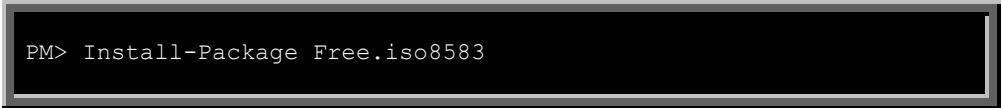
Last Edited: September 10, 2015

Part 1: Simple Example

This tutorial will explain the basic idea of how to create client and server application utilizing Free.iso8583 library. In this example, both applications will communicate each other. This tutorial will not show any Visual Studio image for you. It assumes you have been familiar with Visual Studio.

Let's begin! Open your Visual Studio and create new project named "tutorial". Under solution "tutorial", create 3 new projects: "Models" (Class Library), "Client" (Console Application) and "Server" (Console Application). Remove project "tutorial", we don't need it.

Project "Models" will contain all model classes. The models are placed in a separate project because they are used by both client and server application. As their name, project "Client" is for client application and project "Server" is for server application. Add 2 references to project "Client" and "Server": the first one refers to project "Models" and the second one refers to Free.iso8583.dll library. You may also add the library via "Manage NuGet Packages..", find the package named 'Free.iso8583' or you may also type the following command in the Package Manager Console:



```
PM> Install-Package Free.iso8583
```

Models

We will use message type 0800 for request message. The message will contain 7th field (Transmission date & time), 11th field (Systems trace audit number), 48th field (Additional data) and 70th field (Network management information code). Under project "Models", create a model class for this request message, as follows:

```
public class Request0800
{
    public DateTime TransmissionDateTime { get; set; }
    public int SystemAuditTraceNumber { get; set; }
    public String AdditionalData { get; set; }
    public String NetworkManagementInformationCode { get; set; }
}
```

Code 1. Model Class for Request

Response message (type 0810) will contain 7th field (Transmission date & time), 11th field (System trace audit number), 39th field (Response code), 64th field (Message authentication

code), 48th field (Additional data) and 70th field (Network management information code). Under project “Models”, create a model class for this response message, as follows:

```
public class Response0810
{
    public DateTime? TransmissionDateTime { get; set; }
    public int? SystemAuditTraceNumber { get; set; }
    public String AdditionalData { get; set; }
    public String NetworkManagementInformationCode { get; set; }
    public String ResponseCode { get; set; }
    public byte[] MessageAuthenticationCode { get; set; }
}
```

Code 2. Model Class for Response

Delete class `Class1` that was automatically created under project “Models”, we don’t need it.

Client Application

After constructing models, we have to determine what delegate functions we need. In this case, we need the function for converting `byte[]` to `DateTime`. It’s needed to get the value of property `TransmissionDateTime` that is mapped to 7th field in the message. The format of 7th field is composed of 10 digits of integer (2 digits of month followed by each 2 digits of day, hour, minute and second). Each digit is stored in a nibble. We also need the function to do conversely to convert the value of property `TransmissionDateTime` to the value for 7th field. Fortunately, `Free.iso8583` has provides those functions, those are method

`DateTimeNibble.GetMMDDHHMMSS` and `DateTimeNibble.GetBytesFromMMDDHHMMSS`.

Then, we must create the configuration file. Under root directory of project “Client”, add a new XML file and name it “client-conf.xml”. Add element `MessageMap` as the root element (all other XML elements will reside inside it). Insert two elements below to define the delegate functions as mentioned before.

```
<delegate id="BytesToMmddhmmss" class="Free.iso8583.model.DateTimeNibble"
    method="GetMMDDHHMMSS" param-type="System.Byte[]" />
<delegate id="MmddhmmssToBytes" class="Free.iso8583.model.DateTimeNibble"
    method="GetBytesFromMMDDHHMMSS" param-type="System.DateTime" />
```

Code 3. Delegates Configuration

For request message configuration, insert these XML elements:

```
<message id="req0800" length-header="2">
    <header name="Headers" value="0250000000" length="5" />
    <message-type value="0800" length="2" />
    <bitmap length="16" />
</message>
```

```

    <bit seq="1" type="NULL" />
    <bit seq="7" type="N" length="10" delegate="MmddhmmssToBytes" />
    <bit seq="11" type="N" length="6" />
    <bit seq="48" type="ANS" length-header="2" />
    <bit seq="70" type="N" length="3" />
</message>

<model id="mdlRequest0800" class="Models.Request0800, Models" message="req0800">
  <property name="TransmissionDateTime" bit="7" type="bytes"
    delegate="BytesToMmddhmmss" />
  <property name="SystemAuditTraceNumber" bit="11" type="int" />
  <property name="AdditionalData" bit="48" type="string" />
  <property name="NetworkManagementInformationCode" bit="70" type="string" />
</model>

```

Code 4. Request Message Configuration

You should consult the documentation for detailed explanation about above XML elements and attributes. Here, it will be discussed some of them.

Because there is 70th field, it's needed secondary bitmap. So, the value of attribute `length` of element `bitmap` is 16. It includes 8 bytes primary bitmap and 8 bytes secondary bitmap. The first bit in bitmap must be on to indicate the existence of secondary bitmap. To make the first bit on, we place element `bit` whose `type="NULL"` at the first sequence.

Notice, how to use the delegate we set before. The delegate function which returns `byte[]` is set to element `bit` and the delegate function which takes `byte[]` as the parameter is set to element `property`.

Also notice attribute `class` of element `model` which requires a type name. Different from element `delegate` before, for this attribute we need to write the assembly name after the full name of class. It's because the class is outside of Free.iso8583 and .NET Framework library.

Similar with request message, the configuration for response message is like below.

```

<message id="resp0810" length-header="2">
  <header name="Headers" value="0250000000" length="5" />
  <message-type value="0810" length="2" />
  <bitmap length="16" />
  <bit seq="1" type="NULL" />
  <bit seq="7" type="N" length="10" delegate="MmddhmmssToBytes" />
  <bit seq="11" type="N" length="6" />
  <bit seq="39" type="AN" length="2" />
  <bit seq="48" type="ANS" length-header="2" />
  <bit seq="64" type="B" length="64" />
  <bit seq="70" type="N" length="3" />
</message>

<model id="mdlResponse0810" class="Models.Response0810, Models"
  message="resp0810">
  <property name="TransmissionDateTime" bit="7" type="bytes"

```

```

        delegate="BytesToMmddhmmss" />
        <property name="SystemAuditTraceNumber" bit="11" type="int" />
        <property name="ResponseCode" bit="39" type="string" />
        <property name="AdditionalData" bit="48" type="string" />
        <property name="MessageAuthenticationCode" bit="64" type="bytes" />
        <property name="NetworkManagementInformationCode" bit="70" type="string" />
    </model>

```

Code 5. Response Message Configuration

We also need a callback function to process the response message. Open class `Program` that was created automatically under project “Client”. Don’t forget to include the needed namespaces on the top of code.

```

using System.IO;
using Free.iso8583;
using Models;

```

Code 6. Namespaces needed for `Client.Program`

Add method `ProcessResponse` to be used as the callback function. Write the following code:

```

class Program : IModelParsingReceiver
{
    public Object ProcessResponse(Object model)
    {
        if (model == null)
        {
            Console.WriteLine("An error occurred!! Cannot get response...");
        }
        else
        {
            Console.WriteLine("==== Begin: Response ====");
            Console.WriteLine(
                MessageUtility.HexToReadableString(ParsedMessage.AllBytes));
            Console.WriteLine("");
            Console.WriteLine("");
            Console.WriteLine(Util.GetReadableStringFromModel(model));
            Console.WriteLine("==== End: Response ====");
        }
        return null;
    }

    public IParsedMessage ParsedMessage
    {
        get;
        set;
    }
}

```

Code 7. Client callback function: method `ProcessResponse`

Class `Program` implements interface `IModelParsingReceiver` in order for it gets the result of message parsing including the raw bytes of message that will be set to property

ParsedMessage. Method ProcessResponse above which is used as the callback function only prints the response on the console. The method accepts the response model object as the parameter. This parameter is null if there was an error.

The callback function is also configured as a delegate function. So, insert the configuration as follows:

```
<delegate id="processResponse" class="Client.Program, Client"
  method="ProcessResponse" param-type="System.Object" />
```

Code 8. Configuration for callback function

To tell the message processor which model to be used for the response message, we insert this configuration:

```
<message-to-model model="mdlResponse0810" delegate="processResponse">
  <mask start-byte="8" length="2" value="0810" />
</message-to-model>
```

Code 9. Configuration for mapping incoming message to a model class

Attribute model defines which model class is used and attribute delegate defines the callback function that will be processed when the message comes.

This configuration asks the message processor to check the bytes of MTI (Message Type Indicator). The first byte of MTI is on 8th byte of message. It's counted as follows: the first two bytes are the message length indicator then followed by 5 bytes of header and then MTI (look back at message configuration). The length of MTI is two bytes and its value should be "0810" hexadecimal. All these settings are exposed by element mask above.

To be clearer, here is the complete configuration that was previously written in separate pieces:

```
<?xml version="1.0" encoding="utf-8" ?>
<MessageMap>
  <delegate id="BytesToMmddhhmmss" class="Free.iso8583.model.DateTimeNibble"
    method="GetMMDDHHMMSS" param-type="System.Byte[]" />
  <delegate id="MmddhhmmssToBytes" class="Free.iso8583.model.DateTimeNibble"
    method="GetBytesFromMMDDHHMMSS" param-type="System.DateTime" />

  <message id="req0800" length-header="2">
    <header name="Headers" value="0250000000" length="5" />
    <message-type value="0800" length="2" />
    <bitmap length="16" />
    <bit seq="1" type="NULL" />
    <bit seq="7" type="N" length="10" delegate="MmddhhmmssToBytes" />
    <bit seq="11" type="N" length="6" />
    <bit seq="48" type="ANS" length-header="2" />
    <bit seq="70" type="N" length="3" />
  </message>
</MessageMap>
```

```

</message>

<model id="mdlRequest0800" class="Models.Request0800, Models" message="req0800">
  <property name="TransmissionDateTime" bit="7" type="bytes"
    delegate="BytesToMmddhhmmss" />
  <property name="SystemAuditTraceNumber" bit="11" type="int" />
  <property name="AdditionalData" bit="48" type="string" />
  <property name="NetworkManagementInformationCode" bit="70" type="string" />
</model>

<message id="resp0810" length-header="2">
  <header name="Headers" value="0250000000" length="5" />
  <message-type value="0810" length="2" />
  <bitmap length="16" />
  <bit seq="1" type="NULL" />
  <bit seq="7" type="N" length="10" delegate="MmddhhmmssToBytes" />
  <bit seq="11" type="N" length="6" />
  <bit seq="39" type="AN" length="2" />
  <bit seq="48" type="ANS" length-header="2" />
  <bit seq="64" type="B" length="64" />
  <bit seq="70" type="N" length="3" />
</message>

<model id="mdlResponse0810" class="Models.Response0810, Models"
  message="resp0810">
  <property name="TransmissionDateTime" bit="7" type="bytes"
    delegate="BytesToMmddhhmmss" />
  <property name="SystemAuditTraceNumber" bit="11" type="int" />
  <property name="ResponseCode" bit="39" type="string" />
  <property name="AdditionalData" bit="48" type="string" />
  <property name="MessageAuthenticationCode" bit="64" type="bytes" />
  <property name="NetworkManagementInformationCode" bit="70" type="string" />
</model>

<delegate id="processResponse" class="Client.Program, Client"
  method="ProcessResponse" param-type="System.Object" />

<message-to-model model="mdlResponse0810" delegate="processResponse">
  <mask start-byte="8" length="2" value="0810" />
</message-to-model>
</MessageMap>

```

Code 10. Complete configuration of client application (the content of client-conf.xml)

Now, our task is to write the code to load the configuration and send the request message. These tasks will be done in one method, say it method `Run`. This method is called when the application starts. This is the method `Run`:

```

public void Run()
{
    String configPath = Util.GetAssemblyDir(this) + "/../..\\client-conf.xml";
    try
    {
        MessageProcessor.GetInstance().Load(File.Open(configPath, FileMode.Open));
    }
    catch (FileNotFoundException ex)
    {
    }
}

```

```

    {
        Logger.GetInstance().Write(ex);
        return;
    }

    Request0800 req = new Request0800
    {
        TransmissionDateTime = DateTime.Now,
        SystemAuditTraceNumber = 3456,
        AdditionalData = "Additional Data",
        NetworkManagementInformationCode = "301"
    };

    Console.WriteLine("==== Begin: Request =====");
    Console.WriteLine(Util.GetReadableStringFromModel(req));
    Console.WriteLine("==== End: Request =====");

    try
    {
        MessageClient client = new MessageClient("localhost", 3107, req);
        client.SendModel();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Cannot process the request." + Environment.NewLine
            + ex.Message);
    }
}

```

Code 11. *Method Run of Client.Program*

As seen in the above code, method `Load` of `MessageProcessor` is to load the configuration. This method needs parameter of the configuration file stream. It's assumed the compilation result is placed in directory "bin/Debug" under root directory of project "Client". So, we use the above file path for the configuration.

To send the request message, it calls method `SendModel` of `MessageClient` object. The constructor of `MessageClient` takes three parameters: hostname of server, port of server and request model object.

Ultimately, insert the following code into method `Main`.

```

new Program().Run();

```

Code 12. *Statement to run client application*

Server Application

The main steps to create a server application are similar with a client application's steps. It also needs a method to process the incoming message. Open class `Program` that was automatically

created under project “Server”. Add the method for processing the request message. The method accepts request model object as the parameter and returns response model object. The response model object will be converted to a binary message by the message processor before being sent to the client.

```
public static Response0810 ProcessRequest(Request0800 request)
{
    Console.WriteLine("==== Begin: Request =====");
    Console.WriteLine(Util.GetReadableStringFromModel(request));
    Console.WriteLine("==== End: Request =====");

    Response0810 resp = new Response0810
    {
        TransmissionDateTime = DateTime.Now,
        ResponseCode = "00",
        MessageAuthenticationCode = MessageUtility.StringToHex("0102030405060708")
    };
    return resp;
}
```

Code 13. Server callback function: method *ProcessRequest*

It may need database query and/or some calculations to set the response code and the authentication code. But they’re beyond the scope of this tutorial.

Next, create the configuration file. Add an XML file under root directory of project “Server” named “server-conf.xml”. The complete configuration is similar with the client’s configuration.

```
<?xml version="1.0" encoding="utf-8" ?>
<MessageMap>
    <delegate id="BytesToMmddhmmss" class="Free.iso8583.model.DateTimeNibble"
        method="GetMMDDHHMMSS" param-type="System.Byte[]" />
    <delegate id="MmddhmmssToBytes" class="Free.iso8583.model.DateTimeNibble"
        method="GetBytesFromMMDDHHMMSS" param-type="System.DateTime" />

    <message id="req0800" length-header="2">
        <header name="Headers" value="0250000000" length="5" />
        <message-type value="0800" length="2" />
        <bitmap length="16" />
        <bit seq="1" type="NULL" />
        <bit seq="7" type="N" length="10" delegate="MmddhmmssToBytes" />
        <bit seq="11" type="N" length="6" />
        <bit seq="48" type="ANS" length-header="2" />
        <bit seq="70" type="N" length="3" />
    </message>

    <model id="mdlRequest0800" class="Models.Request0800, Models" message="req0800">
        <property name="TransmissionDateTime" bit="7" type="bytes"
            delegate="BytesToMmddhmmss" />
        <property name="SystemAuditTraceNumber" bit="11" type="int" />
        <property name="AdditionalData" bit="48" type="string" />
        <property name="NetworkManagementInformationCode" bit="70" type="string" />
    </model>
</MessageMap>
```



```

<message id="resp0810" length-header="2">
  <header name="Headers" value="0250000000" length="5" />
  <message-type value="0810" length="2" />
  <bitmap length="16" />
  <bit seq="1" type="NULL" />
  <bit seq="7" type="N" length="10" from-request="true"
    delegate="MmddhmmssToBytes" />
  <bit seq="11" type="N" length="6" from-request="true" />
  <bit seq="39" type="AN" length="2" />
  <bit seq="48" type="ANS" length-header="2" from-request="true" />
  <bit seq="64" type="B" length="64" />
  <bit seq="70" type="N" length="3" from-request="true" />
</message>

<model id="mdlResponse0810" class="Models.Response0810, Models"
  message="resp0810">
  <property name="TransmissionDateTime" bit="7" type="bytes"
    delegate="BytesToMmddhmmss" />
  <property name="SystemAuditTraceNumber" bit="11" type="int" />
  <property name="ResponseCode" bit="39" type="string" />
  <property name="AdditionalData" bit="48" type="string" />
  <property name="MessageAuthenticationCode" bit="64" type="bytes" />
  <property name="NetworkManagementInformationCode" bit="70" type="string" />
</model>

<delegate id="req0800Process" class="Server.Program, Server"
  method="ProcessRequest" param-type="Models.Request0800, Models" />

<message-to-model model="mdlRequest0800" delegate="req0800Process">
  <mask start-byte="8" length="2" value="0800" />
</message-to-model>
</MessageMap>

```

Code 14. Complete configuration of server application (the content of server-conf.xml)

There are some differences you may notice from the above configuration if compared with the client configuration. The first one is the usage of attribute `from-request`. This attribute has the meaning if used in a response message and a server application. If it has `true` value then it means that if not set by the application then the value of the message field should be taken from the request message field on the same sequence number. If you notice the response model class, property `TransmissionDateTime` and `SystemAuditTraceNumber` are made to be nullable. The other properties are class type, so they are nullable. Being nullable is required to make attribute `from-request` works. The value of `null` means it's not set by the application.

The second difference is element `message-to-model` which maps a request message. It's because the incoming message for server application is a request message. It's different from incoming message for client application which is a response message.

Back to the code, we use method `Main` of class `Program` to load the configuration and then start listening to the request.

```
static void Main(string[] args)
```

```

{
    MessageListener messageListener = new MessageListener();

    try
    {
        MessageListener.SetConfigPath("../server-conf.xml",
            Util.GetAssemblyDir(new Program()));
        messageListener.StartListeningEvent += OnStartListeningEvent;
        messageListener.Start();
    }
    finally
    {
        messageListener.Stop();
    }
}

private static void OnStartListeningEvent(Object sender, ListeningEventArgs e)
{
    Console.WriteLine("Starts listening port " + e.Port
        + (e.IPAddress.ToString() != "0.0.0.0"
            ? " on address " + e.IPAddress : "")
        + " ...");
}

```

Code 15. Server application code

Method `Start` of `MessageListener` loads the configuration and then listens to the request. This method blocks as long as the application is running. You must press `Ctrl+C` to stop the application. In the above code, there is an event handler named `StartListeningEvent`. This event is raised when the server starts listening to the request. The handler set above (method `OnStartListeningEvent`) only shows the message on the console which tells that the server is ready to accept the request.

Running the Applications

Consider you place the solution “tutorial” in directory “D:\My Projects”. After compiling the solution, you may find the executable files on path “D:\My Projects\tutorial\Server\bin\Debug\Server.exe” and “D:\My Projects\tutorial\Client\bin\Debug\Client.exe”.

Open a new console window and run the server application. After the server is running, don’t close its window. Open a new window then run the client application.

After communicating, the output of server application should be like this:

```

D:\>My Projects\tutorial\Server\bin\Debug\Server.exe
Starts listening port 3107 ...
==== Begin: Request ====
{
TransmissionDateTime = 10/7/2013 11:00:30 AM
SystemAuditTraceNumber = 3456

```

```
AdditionalData = Additional Data
NetworkManagementInformationCode = 301
}
==== End: Request ====
```

Figure 1. Output of server application

The output of client application should be like this:

```
D:\>My Projects\tutorial\Client\bin\Debug\Client.exe
==== Begin: Request ====
{
TransmissionDateTime = 10/7/2013 11:00:30 AM
SystemAuditTraceNumber = 3456
AdditionalData = Additional Data
NetworkManagementInformationCode = 301
}
==== End: Request ====
==== Begin: Response ====
00 3C 02 50 00 00 00 08 10 82 20 00 00 02 01 00 01 04 00 00 00 00
00 00 00 10 07 11 00 31 00 34 56 30 30 00 15 41 64 64 69 74 69 6F
6E 61 6C 20 44 61 74 61 01 02 03 04 05 06 07 08 03 01

{
TransmissionDateTime = 10/7/2013 11:00:31 AM
SystemAuditTraceNumber = 3456
AdditionalData = Additional Data
NetworkManagementInformationCode = 301
ResponseCode = 00
MessageAuthenticationCode = 01 02 03 04 05 06 07 08
}
==== End: Response ====
```

Figure 2. Output of client application

NOTE:

- Loading configuration should be executed only once for entire lifecycle of application to avoid decreasing performance due to XML parsing process. After that, it may send/receive request/response message more than once.
- You don't have to use class `MessageClient` and/or `MessageListener` included in `Free.iso8583.dll`. You may create new class which is more suitable for your application as long as the class implements interface `IMessageStream`. Please learn the code of `MessageClient` and `MessageListener` for the basic idea.

Part 2: Web-based Application (Another Simple Example)

It's impossible to create a web-based application for ISO 8583 server, except for *back office* only, because the server must give a response in format of ISO 8583 data protocol, not HTTP.

However, it's possible to create a web application which acts as an ISO 8583 client. In common use, a web application is an application that runs on a web server. It's also true for ASP.NET application we usually create. Therefore, a web application is a server application. It serves the request from browser (Internet Explorer, Firefox, Chrome, Opera etc.). So, if it's a server application then how ISO 8583 client works. Take a look at the schema below.

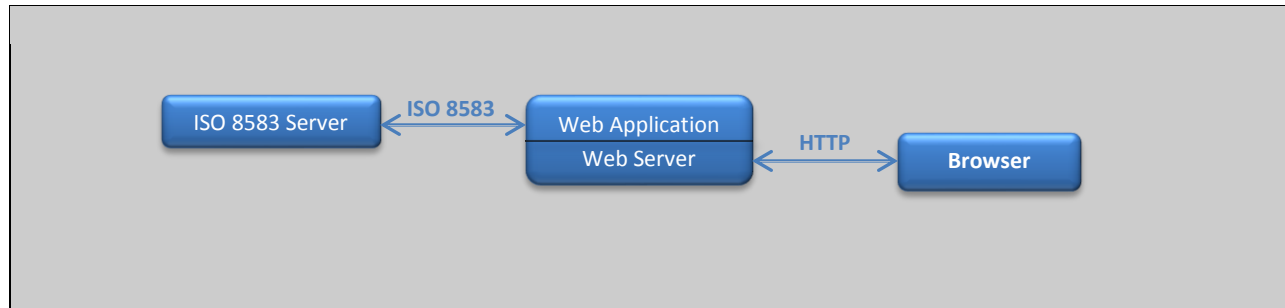


Figure 3. Web-based application architecture for ISO 8583 client

The web application will serve the request from a browser and then it makes a request to an ISO 8583 server based on the parameter inputted via browser. The response data/status from ISO 8583 server will be returned to the browser.

In most server environment, each request-response is stateless. It means that all objects created during request-response cycle will be destroyed after the end of that cycle. It also means that a request cannot make a reference to the previous request. It's for memory usage efficiency. However, there is usually a place to share some objects for some requests. Specifically for web application, there is *cookie/session* to share information among consecutive requests from the same client.

A script that we usually create for a web application is to serve the request. In other words, it will be executed every time a request comes. In the case of Fee.iso8583, we need to load the configuration only once when the application starts, not every time a request comes. Even though it's also possible to load the configuration every time a request comes but it's not efficient because of performance impact. So, is there a place for us to put some initialization codes when the application starts? Fortunately, ASP.NET provides that place. If you open file Global.asax.cs, there are some event handlers. One of them will be executed when the application starts.

To be clearer, let's create an example of web application. Under solution "tutorial" you have created before in tutorial part 1, add new project (ASP.NET Web Application Visual C#). Name it "WebClient". Assumed, we use the same ISO 8583 server that we have created in tutorial part 1. We will also use the same configuration as used by project "Client". Add the references to project "WebClient" which refer to project "Models", project "Client" and Free.iso8573.dll library. Make sure the platform target of project "Client" is 'Any CPU' to avoid error when starting WebClient.

Open file Global.asax.cs and add some statements to load configuration in `Application_Start` event handler. Look at the following code.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;
using Free.iso8583;

namespace WebClient
{
    public class Global : System.Web.HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            String rootDir = Server.MapPath("~/");
            MessageProcessor.GetInstance().Load(File.Open(rootDir
                + "../Client/client-conf.xml", FileMode.Open));
        }

        void Application_End(object sender, EventArgs e)
        {
        }

        void Application_Error(object sender, EventArgs e)
        {
        }

        void Session_Start(object sender, EventArgs e)
        {
        }

        void Session_End(object sender, EventArgs e)
        {
        }
    }
}

```

Code 16. *Global.asax.cs*

Next step, add a Web Form and name it "RequestForm". This web page will be used as an input form to fill the parameters for an ISO 8583 request. Then it will make the ISO 8583 request and show its response. Open RequestForm.aspx and open the Source tab. Replace all code inside there with the following code:

```

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="RequestForm.aspx.cs"
Inherits="WebClient.RequestForm" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>ISO 8583 Testing Form</title>
    <script type="text/javascript">
        function cancelNonNumricKey(e) {
            if (!e) e = window.event;

```

```

        var iKeyCode = e.charCode ? e.charCode : e.keyCode;
        if (48 <= iKeyCode && iKeyCode <= 57) return; //Numeric keys
        if (iKeyCode == 8) return; //backspace
        if (e.charCode === 0) { //Firefox
            if (iKeyCode == 37 || iKeyCode == 39
                || iKeyCode == 46) return; //left arrow, right arrow, delete
        }
        if (e.preventDefault) {
            e.preventDefault();
            e.stopPropagation();
        } else {
            e.returnValue = false;
            e.cancelBubble = true;
        }
    }
}
</script>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Panel ID="pnlInput" runat="server">
            <table border="0">
                <tr>
                    <td align="right" nowrap="nowrap">System Audit Trace Number:</td>
                    <td><asp:TextBox ID="txtTraceNumber" runat="server" MaxLength="6"
Columns="8" onkeypress="cancelNonNumricKey(arguments[0])"></asp:TextBox></td>
                </tr>
                <tr>
                    <td align="right" valign="top" nowrap="nowrap">Additional Data:</td>
                    <td><asp:TextBox ID="txtAdtData" runat="server" Rows="5" Columns="50"
TextMode="Multiline" Wrap="true"></asp:TextBox></td>
                </tr>
                <tr>
                    <td align="right" nowrap="nowrap">Network Management Information
Code:</td>
                    <td><asp:TextBox ID="txtNMICode" runat="server" MaxLength="3" Columns="5"
onkeypress="cancelNonNumricKey(arguments[0])"></asp:TextBox></td>
                </tr>
                <tr>
                    <td>&nbsp;</td>
                    <td><asp:Button ID="btnSubmit" runat="server" Text="Submit"
onclick="btnSubmit_Click" /></td>
                </tr>
            </table>
        </asp:Panel>

        <asp:Panel ID="pnlResponse" runat="server" Visible="false">
            <pre runat="server" id="iso8583Data"></pre><br />
            <asp:LinkButton ID="linkBack" runat="server"
onclick="linkBack_Click">Back</asp:LinkButton>
        </asp:Panel>
    </form>
</body>
</html>

```

Code 17. RequestForm.aspx

It's not in the scope of this tutorial to explain what the really meaning of the above code is. This tutorial assumes you have understood ASP.NET sufficiently. Briefly, there are two panels: the first one contains the input form and the second one is to show the ISO 8583 data. Firstly when

the web page is loaded in the browser, the first panel is displayed and the second one is hidden. After submitting the form, the second panel is displayed and the first one is hidden.

Next, open RequestForm.aspx.cs and replace all code there with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using Free.iso8583;

namespace WebClient
{
    public partial class RequestForm : System.Web.UI.Page, IModelParsingReceiver
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            pnlInput.Visible = Session["Free.iso8583.Data"] == null;
            pnlResponse.Visible = Session["Free.iso8583.Data"] != null;
            if (Session["Free.iso8583.Data"] != null)
            {
                iso8583Data.InnerText = Session["Free.iso8583.Data"].ToString();
            }
        }

        private System.Threading.AutoResetEvent _iso8593ResponseComesEvent
            = new System.Threading.AutoResetEvent(false);
        private String _output = "";
        protected void btnSubmit_Click(object sender, EventArgs e)
        {
            Models.Request0800 req = new Models.Request0800
            {
                TransmissionDateTime = DateTime.Now,
                SystemAuditTraceNumber = int.Parse(txtTraceNumber.Text.Trim()
                    .PadLeft(1, '0')),
                AdditionalData = txtAdtData.Text,
                NetworkManagementInformationCode = txtNMICode.Text.PadLeft(3, '0')
            };

            _output = "==== Begin: Request =====" + Environment.NewLine
                + Util.GetReadableStringFromModel(req) + Environment.NewLine
                + "==== End: Request =====" + Environment.NewLine;

            MessageClient client = new MessageClient("localhost", 3107, req);
            client.Callback = this.PrintIso8583Response;
            client.SendModel();
            _iso8593ResponseComesEvent.WaitOne();

            Session["Free.iso8583.Data"] = _output;
            Response.Redirect("~/RequestForm.aspx");
        }

        private Object PrintIso8583Response(Object model)
        {
            if (model == null)
            {
                _output += "An error occurred!! Cannot get response...";
            }
        }
    }
}
```

```

    }
    else
    {
        _output += "==== Begin: Response ====" + Environment.NewLine
            + MessageUtility.HexToReadableString(ParsedMessage.AllBytes)
            + Environment.NewLine
            + Environment.NewLine
            + Environment.NewLine
            + Util.GetReadableStringFromModel(model) + Environment.NewLine
            + "==== End: Response ====";
    }
    _iso8593ResponseComesEvent.Set();
    return null;
}

protected void linkBack_Click(object sender, EventArgs e)
{
    Session.Remove("Free.iso8583.Data");
    Response.Redirect("~/RequestForm.aspx");
}

//Implements IModelParsingReceiver
public IParsedMessage ParsedMessage
{
    get;
    set;
}
}
}

```

Code 18. RequestForm.aspx.cs

The ISO 8583 request is done after clicking Submit button. So, `btnSubmit_Click` event handler does an ISO 8583 request. There are some notes that may need your attention from the above code:

- Setting property `client.Callback` overrides the callback function that has been set in the configuration. So here, we will use method `PrintIso8583Response` as the callback function instead of method `Client.Program.ProcessResponse`.
- The use of an `AutoResetEvent` object above is to make sure that the callback function (method `PrintIso8583Response`) is executed completely before continuing because method `SendModel` is asynchronous. If not waiting the callback function done, it's possible that the callback function is called after the corresponding ASP.NET request-response cycle has ended. In that situation, the callback function becomes to have no meaning. So, it's important to break the process of event handler until the callback function is executed completely.
- At the end of `btnSubmit_Click`, there are two lines:

```

Session["Free.iso8583.Data"] = _output;
Response.Redirect("~/RequestForm.aspx");

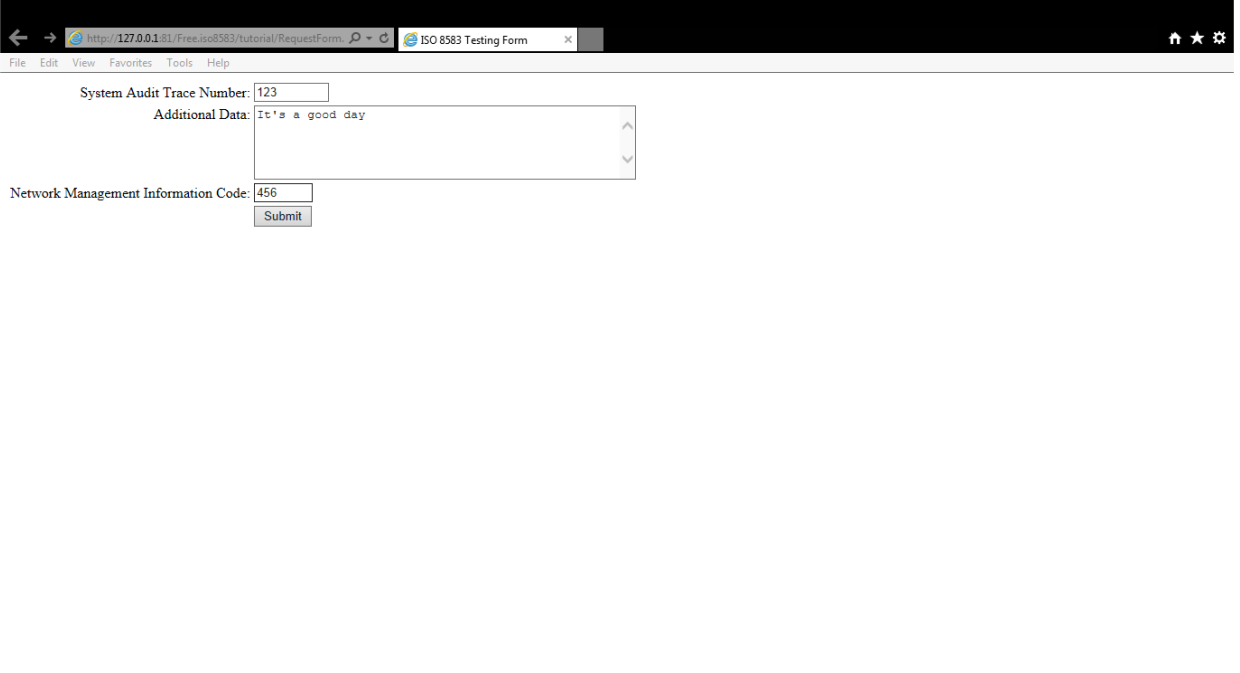
```

Here, we don't show the ISO 8583 data directly but store it in the session storage and reload the web page. After the page is reloaded, the ISO 8583 data stored previously in the session storage is displayed. It's a trick to avoid the message like "the web browser needs to resend

the information you've previously submitted" when we try to refresh the browser after submitting the form. In `Page_Load` event handler, the existence of this session data is checked to determine which panel to be displayed. If exists, this session data is shown.

- Don't forget to implement interface `IModelParsingReceiver` if we want to get the ISO 8583 raw message.

Next step, open Internet Information Service (IIS) Manager and add an Application which refers to project "WebClient". Rebuild the solution. Run the ISO 8583 server (project "Server"). Open a browser and load the web page we have created (`RequestForm.aspx`). The view of the web page should be like these two images below:



The screenshot shows a web browser window with the title "ISO 8583 Testing Form". The address bar displays the URL "http://127.0.0.1:81/Free.iso8583/tutorial/RequestForm.aspx". The browser's menu bar includes "File", "Edit", "View", "Favorites", "Tools", and "Help". The main content area of the browser displays a web form with the following elements:

- A label "System Audit Trace Number:" followed by a text input field containing the value "123".
- A label "Additional Data:" followed by a text area containing the value "It's a good day".
- A label "Network Management Information Code:" followed by a text input field containing the value "456".
- A "Submit" button located below the "Network Management Information Code" input field.

Figure 4. *Input Form*

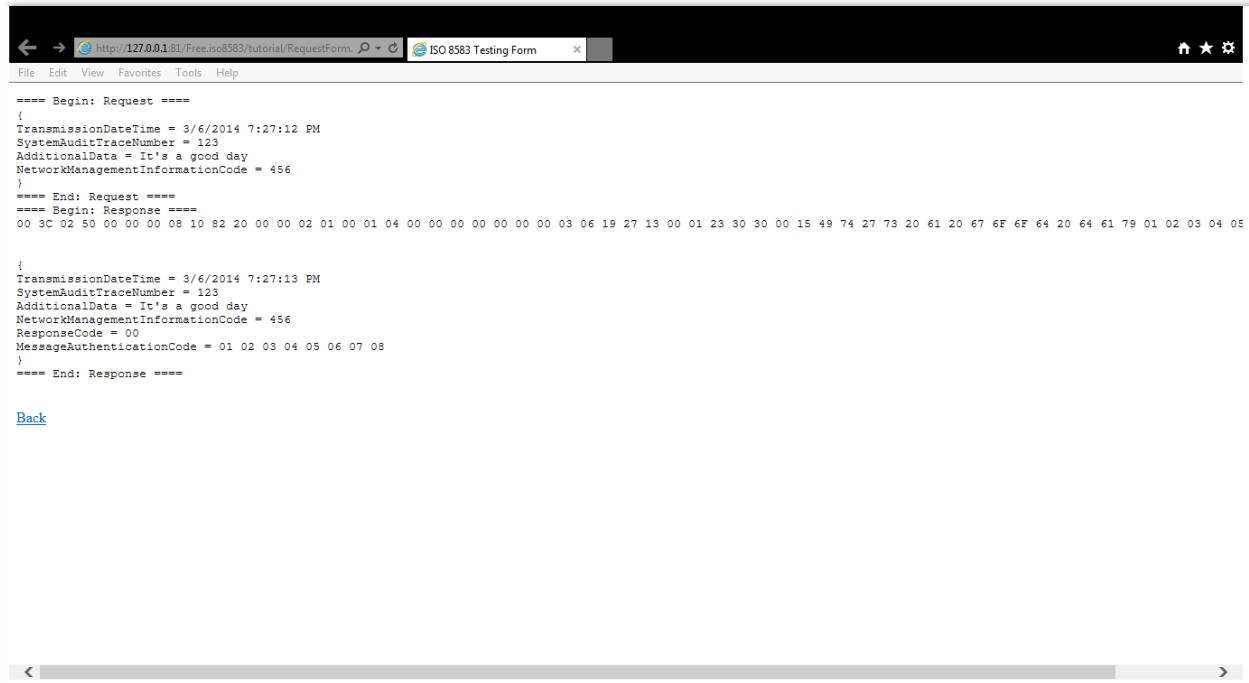


Figure 5. The output of ISO 8583 data

The last question: Do Free.iso8583 configuration objects become the shared objects for all web requests in this web application because we load the configuration only once when the application starts? Yes, they do. Try to submit the form several times with different input. You'll see it works.

DONATION

If you feel this library is useful, please make a donation via PayPal by clicking this button/link:



https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=4ZKPC3URPZ24U