Project Mockingbird

Installation & User Guide - (For V2.0 RTM)

Santosh Benjamin

Installation & User Guide

Table of Contents

Revision History

Version	Summary of changes	Author	Date
1.0a	Alpha release	Santosh Benjamin	13-Jan-2009
1.0b	Beta release	Santosh Benjamin	06th May 2009
1.0c	Beta -2 release	Santosh Benjamin	18th May 2009
1.0d	V1 RTM release	Santosh Benjamin	21st Nov 2009
2.0a	V2 RC release	Santosh Benjamin	17th Mar 2010
2.0 – RTM	V2 RTM release	Santosh Benjamin	12th Aug 2010

Introduction

This document explains how to setup the various components of MockingBird

Components of MockingBird

The components of mockingbird are as follows

- An ASP.NET HttpHandler
- A WCF message interceptor
- A WCF service host
- A simulation framework
- Engine configuration files
- An XML Instance Generation library
- A 'Service Studio application that has three main modules
    - A module to retrieve metadata from a supplied module and generate a default mock service.
    - A module to parse a given WSDL and generate sample requests and responses and help in setting up a mock service quickly.
    - A module to invoke the test services or remote services and check the responses.

Installation Packages & Physical Layout

In the v2.0 RC release, there is a single MSI that deploys a web application and a separate MSI that deploys the Service Studio GUI.

In the v2.0 RTM release, there are 2 MSIs for the Simulator Service (one that deploys the web application and one that deploys the Console Host) and another MSI for the Service Studio WinForms app.

In terms of the physical layout MockingBird is manifested as follows

- A virtual directory : This contains the SimulatorService.svc file (the WCF message interceptor) and in the bin folder we have the Interceptor.dll which contains HttpHandler (implemented as a class that inherits the IHttpHandler interface rather than an .ashx). The .svc and the httpHandler both share the web.config and as explained in the technical documentation, there are a number of other .config files
- A folder (usually installed into Program Files) containing the Console Host application and a testing app (also a console that fires off preset messages to the test endpoints that are hosted in the console app)
- A folder (usually installed into Program Files) containing the Service Studio GUI application.
- The datafiles folder containing a **configuration** subfolder and **endpoints** subfolder. This is available in both the web app install folder and the console app install folder.

Installation Guide

This is a short description of what you need to get up and running with the tool. A detailed description of the components (and the folders needed etc) can be found in later sections of the document. The next two sections will explain how to test your installation and how to deploy your mock endpoints.

Get & Install the package

(1) Run the setup of the web application which will create a virtual directory (by default). If you wish you can create a separate website and application pool but by default the system will install mockingbird into the defaultwebsite and use the DefaultAppPool.

(2) Run the setup of the console host application.

(3) Run the setup of the service studio.

(4) Extract the datafiles..(Sample path: c:\inetpub\wwwroot\MockingBird\datafiles). This data files folder also contains a 'sample' end point which is required to verify the installation (and the verification is done using the Studio app). Once you are proficient with this app and don't need it, you can delete this sample folder if necessary.

(5) From a Powershell prompt (in the MockingBird directory) run the **grantPermissions.ps1** script. This will give full control access to "Everyone" on the MockingBird data files folder. If you are not comfortable with giving such sweeping permissions, then feel free to change the script or manually ensure you give access to Network service or ASPNET user on the mockingbird datafiles folder (Even the service studio user needs permission to these folders as it writes entries to the handlerMap.config file).

Please note that it is preferable **NOT TO** run both source and binary versions on the same box because that can cause a lot of confusion over which website is actually being called by the consumer and where the configuration files are located. If you are changing the code of mockingbird then i recommend you treat it as a regular development exercise so you develop with WebDev locally and deploy to IIS on your central server (whatever that is).

Managing Configuration

We don't expect that you will install and use both the web app and the console host. However, if you do have this scenario then you need to manage configuration carefully.

To keep things simple, we have opted to package the "datafiles" folder with both Simulator MSIs (that is, the web app as well as the console app). The datafiles folder contains the **configuration** folder with all the config files as well as the **endpoints** folder where you create new subfolders for each endpoint that you are mocking.

You can choose to use either location (the datafiles in the web app folder or the datafiles in the console app folder) or choose to use keep your web app config and console app config completely separate.

The simplest scenario, especially if you are not familiar with MockingBird, is to just setup the web app and the datafiles folder. Then if you want the additional protocols support from the console app, set that up and make the .exe.config file point to the configuration folder that sits with the web.app.

However you choose (to centralize the config or keep them separate) you will need to update the web.config and exe.config (as well as the **mockingbird.config)** file so that the apps know where to look for their configuration.

If centralized configuration management via a UI is something you see as becoming essential then please submit a feature request for the same. For the present, we wanted to keep it really simple.

Also note that the config files have automatically been populated with the common settings/paths so if you just use all the defaults (install locations etc) then you probably wont need to touch anything at all in the config files.

Update the configuration files

(1) **Web.config** file: The only setting in this file is "appSettings" **mockingbird_configuration_folder.**

```
<appSettings>
  <add key="mockingbird_configuration_folder" value="C:\inetpub\wwwroot\mockingbird\datafiles\configuration"/>
</appSettings>
```

(2) **Log4Net Config File** : This is in the **configuration** folder (subfolder of **datafiles).** If you are using Log4Net, then modify this file. You can change the path to the runtime logs here or change the entire appender if you are familiar with log4net (such changes are outside the scope of this document). Ensure that the debuglog.txt (the output file) is not read-only as log4net just fails silently if the file is read-only.

```
<appender name="LogFileAppender" type="log4net.Appender.RollingFileAppender">
  <param name="File" value="C:\MyProjects\CodePlex\MockingBird\main\datafiles\logs\runtime\debuglog.txt" />
  <param name="AppendToFile" value="true" />
```

(3) **EntLib Logging Config File** : This is in the **configuration** folder (subfolder of **datafiles).** If you are using EntLib then modify this or else ignore it. As shown below there are two flat file trace listeners (actually pointing to the same file) configured and also an event log listener. Additionally there is a catch-all event log sink in case there are errors writing to the logs or if there are categories used (in case the source is modified) where there is no sink defined. Again, ensure that the log output file is not read only.

```
<listeners>
  <add fileName="C:\MyProjects\CodePlex\MockingBird\main\datafiles\logs\runtime\debuglog.txt"
    header="----------------------------------------" footer="------------------------------
    formatter="Text Formatter" listenerDataType="Microsoft.Practices.EnterpriseLibrary.Loggir
    traceOutputOptions="None" filter="All" type="Microsoft.Practices.EnterpriseLibrary.Loggir
    name="FlatFile TraceListener" />

  <add fileName="C:\MyProjects\CodePlex\MockingBird\main\datafiles\logs\runtime\debuglog.txt"
    formatter="SimpleTraceFormatter" listenerDataType="Microsoft.Practices.EnterpriseLibrary.
    traceOutputOptions="None" filter="All" type="Microsoft.Practices.EnterpriseLibrary.Loggir
    name="Simple FlatFile TraceListener" />

  <add source="MockingBird.Interceptors.Classic" formatter="Text Formatter"
    log="Application" machineName="" listenerDataType="Microsoft.Practices.EnterpriseLibrary.
    traceOutputOptions="None" filter="All" type="Microsoft.Practices.EnterpriseLibrary.Loggir
    name="Formatted EventLog TraceListener" />
```

(4) **MockingBird.Config file** : This is in the **configuration** folder (subfolder of **datafiles).** This configuration file is described in detail later, but the two sections to modify are shown below.

a. The first section to modify is the “Global” settings which has folder locations that need to be set to the “datafiles” which were unzipped in the previous steps. There is also a setting pointing to the location of the configuration XSD which by default is in the “bin” folder of the web application.

```
<Global>
  <Setting name ="AppRoot" value="C:\inetpub\wwwroot\MockingBird" />
  <Setting name ="DataFilesRoot" value="$(AppRoot)\Datafiles" />
  <Setting name ="EndpointsRoot" value="$(DataFilesRoot)\endpoints" />
  <Setting name ="LogFilesRoot" value ="$(DataFilesRoot)\logs"/>
  <Setting name ="ConfigSchemaFilesLocation" value="$(AppRoot)\bin\" />
  <Setting name="TimeoutForce" value="10000" />
</Global>
```

(5) **Service Studio app.config file**: (This is usually located in c:\program files\mockingbird\mockingbird.servicestudio\). The path to the configuration fils as well as the settings unique to the studio need to be changed here. Some of the key settings are the URI to the **endpoint** (if you are using WebDev, don’t forget to specify the port as well), the default soapAction for the test message and the default location of the TestData files.

```
<appSettings>
  <!--Root configuration folder. This is a Vital key.-->
  <add key="mockingbird_configuration_folder" value="c:\inetpub\wwwroot\mockingbird\datafiles\configuration"/>

  <!-- MODULE settings-->
  <!-- Settings for the Service Configurator-->
  <add key ="default_mb_root" value="c:\inetpub\wwwroot\mockingbird"/>

  <!-- Settings for the Service Invoker-->
  <add key="default_asmx_url" value="http://localhost/mockingbird/"/>
  <add key="default_wcf_url" value="http://localhost/mockingbird/SimulatorService.svc/"/>
  <add key="default_endpoint_name" value="sample.asmx"/>
  <add key="default_soapaction" value="http://www.contoso.com/:GetPerson"/>
  <add key="default_req_msg_location" value="c:\inetpub\wwwroot\mockingbird\datafiles\endpoints\sample\requests

  <!-- Settings for the Message Instance Builder-->
  <add key="default_wsdl_location" value="C:\Program Files\MockingBird\ServiceStudio\TestData\MathService_Monol
```

In this version, the uniqueness of the end point is established by the last segment of URI (eg: customer.asmx rather than the whole http URL with port numbers etc) So if you are testing several endpoints, change the last segment of the URI.

Do note that for v2 RTM, the studio can only test the console app endpoints that use basicHttpBinding as it does not have any client.config that can be used to test other bindings..

Testing your installation

The Service Studio application can help you verify your installation as well as rapidly create new mock services.

Testing your installation using the Service Invoker

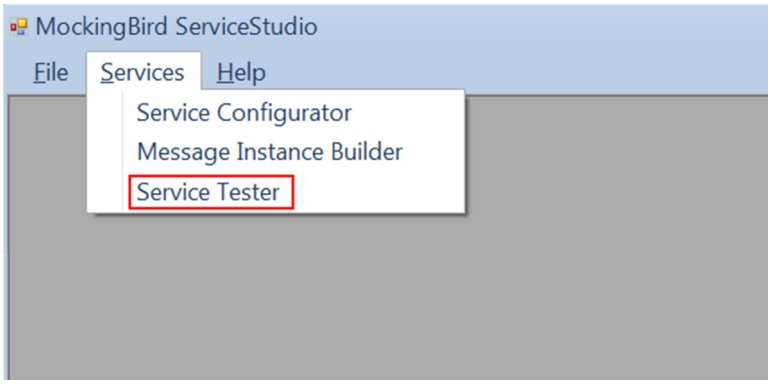
The Service Studio application **ServiceInvoker** module is useful for verifying that the install is working correctly and can also be used to test all your mock endpoints (It can be used to test any other SOAP endpoint as well however at this time the design and settings are focussed towards testing the mock services.)

For verification of the installation, this app requires the "sample" endpoint to be available in the datafiles\endpoints folder. (It should be available by default anyway, as its included in the package).

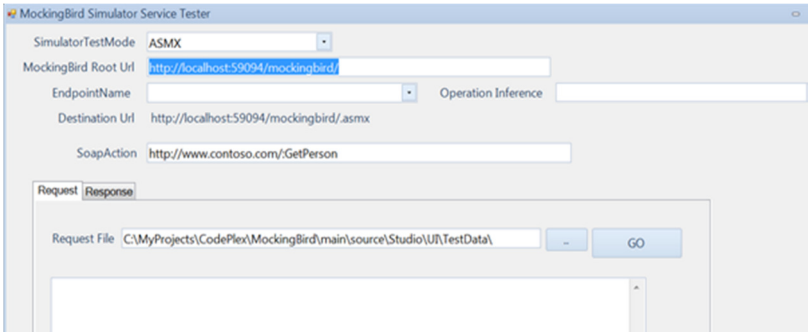
The Invoker can test both the HttpHandler as well as the WCF message interceptor (in Http Mode). To test the WCF simulator Host (non Http) a separate simulator tester application is provided and is co-located with the simulator host in the web app bin folder. There is a known issue that the "Service Tester" console app doesn't work in this version. It was meant to test all the other bindings (other than basicHttp). In a future release the Service Tester console app functionality will be merged into the studio so it becomes a one stop shop

(1) Launch the application – the parent MDI window is shown. The tools are under the Services menu.

(2) To Test the service invoker, select the Service Tester option from the Services Menu as shown in the figure below.

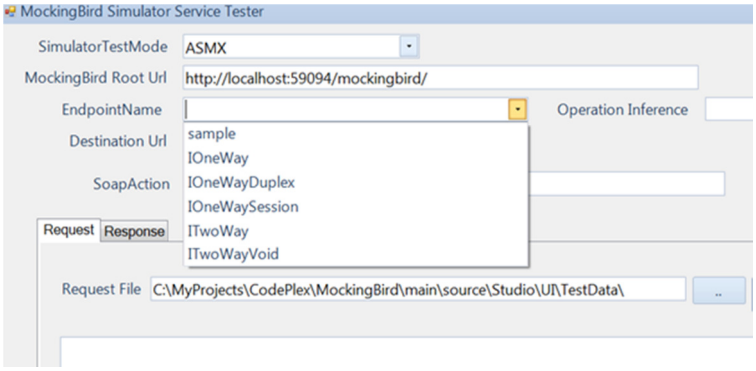


(3) The Service Studio form opens up as shown in the following figure.



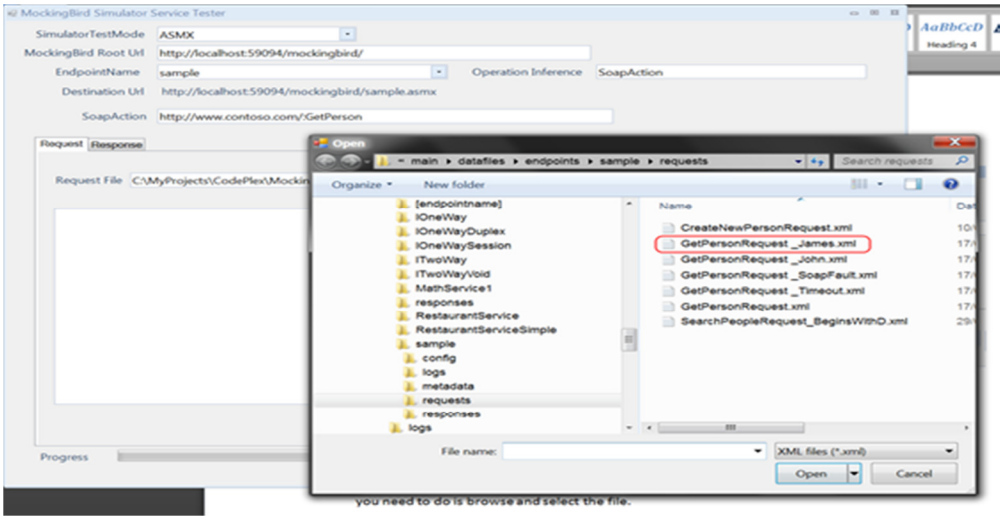
As explained earlier, this module has now been focussed on testing the mock service, in particular the ASMX handler (although we can also test any endpoints hosted by the Simulator Console Host as long as they support basicHttpBinding.

(4) The Endpoint Name dropdown is populated from the datafiles\ configuration\ **handlerMap.config** file automatically (shown in the figure below).

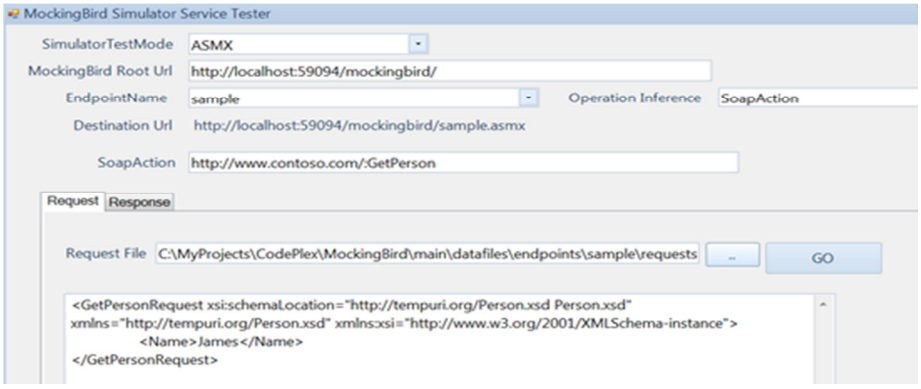


(5) Now select the **sample** endpoint from the dropdown. The system automatically sets the endpoint name (http://localhost/mockingbird/sample.asmx) and it also picks up the Operation Inference strategy that was set in the config file for that endpoint. In the case of the default endpoint, we have already set it to SoapAction, so the corresponding soap action for the Get operation is set and the path to the **request** files has also been initialized so all you need to do is browse and select the file.

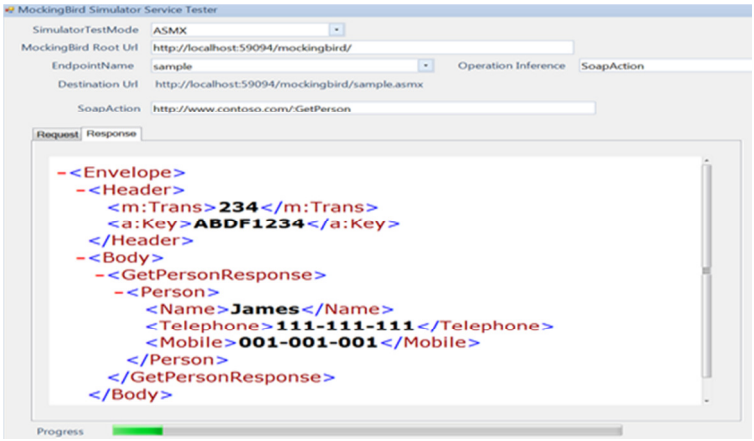
Pick any of the Get requests (for example : GetPerson_James) as shown in the figure below.



(6) The request is now loaded into the request box as shown below



(7) Now click GO. The system will create a soap request using the request message shown and will post it to the endpoint (in the figure the URI is http://localhpst:59094/mockingbird/sample.asmx)
The focus is then automatically set to the Response tab and the response from the service is shown .



In general you need to ensure that if you edit the request, then the XML is well-formed. The system does not do any validation in this version so any errors will be whatever is thrown by the framework. You will need to change the soapAction if you ar etrying out the CreatePerson test

Creating Mock Services using the Configurator

Important Note – HandlerMap.config

The configurator module described below will create the service for you and add an entry into the handlerMap.config file.

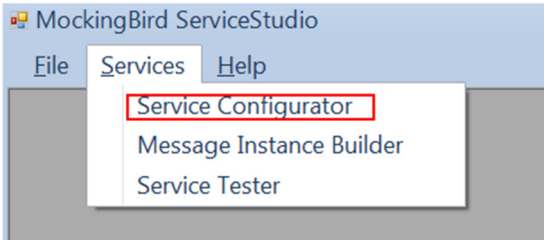
For the ASMX service this is enough because in web.config we specify that we want to intercept all requests (for .asmx) and then when we get the request , we look into handlerMap.config to find if the endpoint is registered there (and in doing so, we also ignore the .asmx extension).

However, for the WCF service, an entry in the handlerMap alone is not enough. The web.config (and the console host .exe.config) needs to hold all the endpoints that the WCF service is listening on. (It is not as easy as doing a *.asmx) so in the current version you need to MANUALLY add an entry to the web.config (and/or the .exe.config) so that the WCF service is aware of the new endpoint. In the next version we will look into automating this.

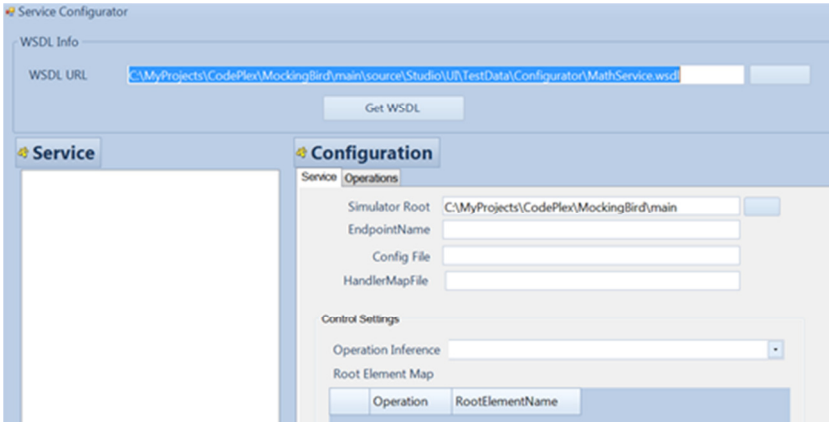
Generating the Service

The configurator module allows you to generate a complete mock service with only a few clicks. In this version it only generates all defaults and does not allow full customization of the response handling etc. Richer UI support will be available in a future version.

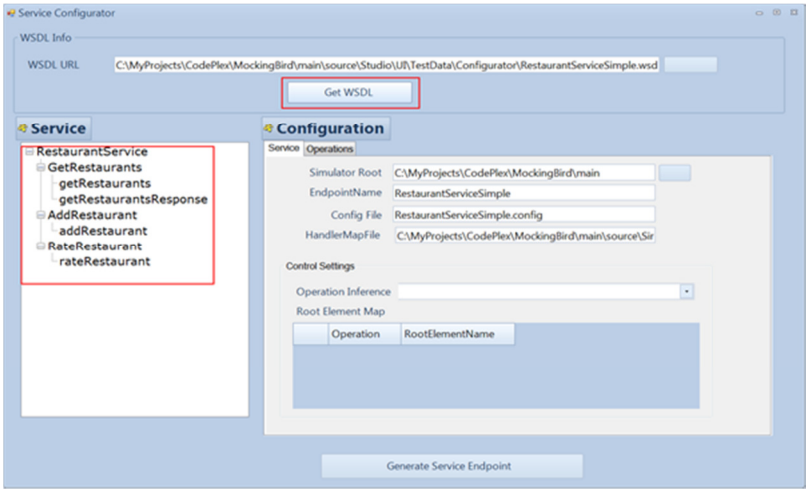
(1) Launch the Service Configurator module from the main Studio menu.



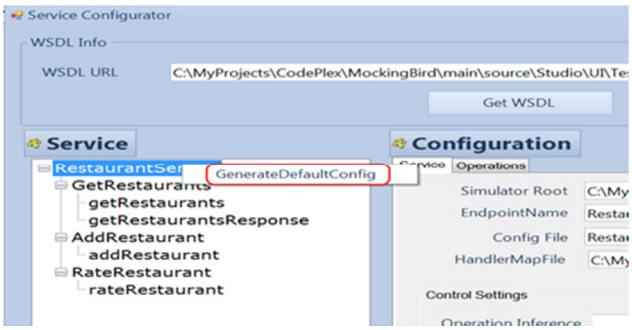
(2) The module is launched with a default WSDL Url shown. There are two test WSDLs (RestaurantServiceSimple and MathService provided with the application so that you can test out the mock service generation process. If you have the WSDL locally then browse to the path on file or else enter in the URL of the 'live' service (or the metadata endpoint) and the system will automatically pick up the metadata



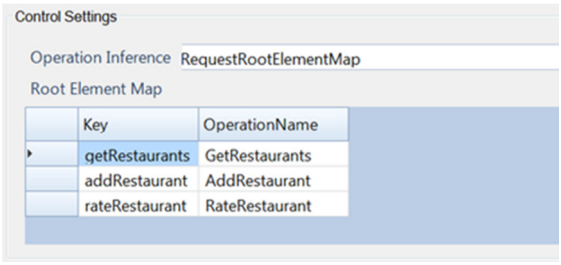
(3) Now click on GetWSDL (in the screen below we have selected the RestaurantService sample WSDL). The system retrieves the metadata and shows the structure in the Service treeview. Some default settings are also immediately populated and shown in the Configuration section.



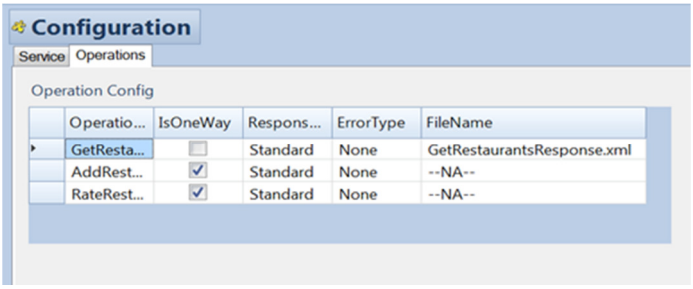
(4) Now right click on the top level node in this case “RestaurantService” and a context menu is shown



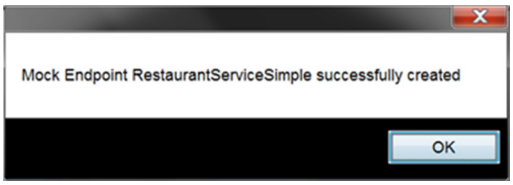
(5) Now click the option shown in the context menu - GenerateDefaultConfig. The system generates some control settings (RequestRootElementMap) and shows how the request root elements map to the operations.



In the operations tab, it also shows a list of all the operations and the response file names (if the operation is 2 way) . In the fig below only GetRestaurants is 2 way.



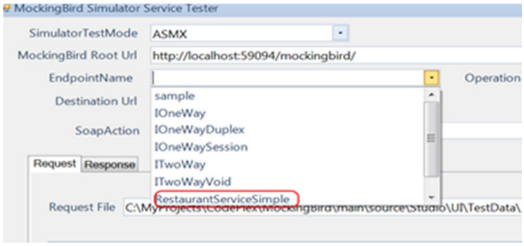
(6) All that remains is to click the button “Generate Service Endpoint” at the bottom of the screen. The system then generates a new mock service folderset in the datafiles\endpoints\(servicename) as well as the config for the service.



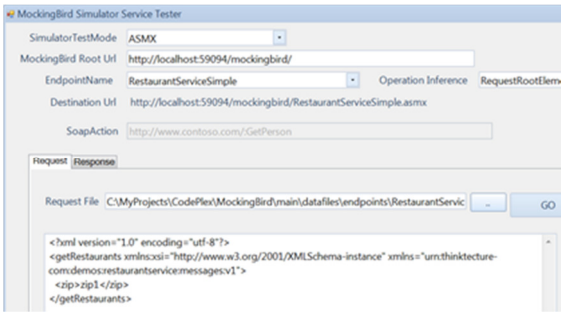
Testing the new mock service

Now that you have generated a mock service successfully, you can test it using the Service Tester module.

(1) When you launch the Service Tester , the new RestaurantServiceSimple endpoint is available in the dropdown.



(2) Pick the RestaurantServiceSimple endpoint . The system automatically sets the destination URL, Operation Inference, paths to the request file etc (as described earlier). Pick the "getRestaurants" request message.

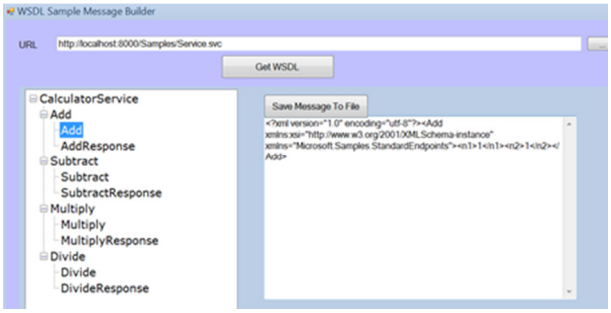


(3) Click GO and the response is obtained from the mock endpoint.

Using Message Instance Generator

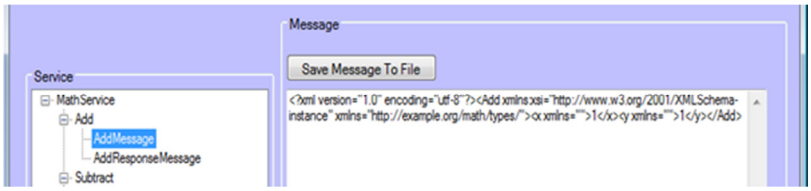
While the Service Configurator can setup all the default request and response messages for a service endpoint, as the service scope grows you will need to be setting up your own sample messages. The message instance generation module aims to simplify this by allowing you to pick the operation message and then select the action to generate a sample message. This is a fairly simple implementation of message instance generation and will always generate the same message. You can edit and save the file. If you are keen to have more functionality in this area, let us know and we can enhance it in future versions.

(1) Select the Message Instance Generator option from the Services menu. The Instance Generator form location. Click on the GET WSDL button and the WSDL is parsed and displayed in the tree view as shown in the following screenshot. Note that in the screen below, we have used the standard Calculator service that comes with the WF_WCF_Samples and this also serves to illustrate that the metadata parsing capability of the system includes standard WCF Multipart wsdl.



(2) Now right click on any of the message nodes (eg: AddMessage or AddResponse message) and a context menu is displayed. Currently this menu has only one option, namely – Generate Sample as shown in the screenshot below.

(3) Now when you click the Generate Sample menu, a sample message is generated and populated on the Message text box to the right of the tree.



(4) The message is left editable so that we can create more responses simply by editing the first sample instead of regenerating each time. Click on the Save Message To File button to save the response. The save file dialog does not have some of the standard file filters set, but this will be fixed soon. However the file can still be saved as an XML File. Note that no validation is done to check if it is well formed or valid prior to saving. Again this is something that can be fixed by the final release.

Manually Creating Mock Services

Mock service creation can be a fairly laborious process for a complex WSDL (with many operations) so the Service Configurator module can be invaluable for such scenarios and we would recommend using that for any new mock service you create. Once the default service is created, then you can edit the configuration to specify different types of responses and add more sample response messages to the response folder.

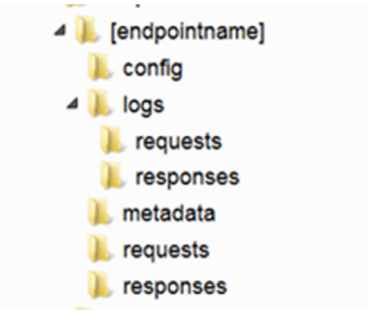
If you do choose to manually setup a new service, then the key things to remember are

- Setup the endpoint folder
- Create a new entry in the handlerMap.config file (for both ASMX and WCF endpoints)
- For WCF you also need to update the web.config and/or the Console Host .exe.config with the endpoint/binding/contract information.

In a future version we will externalize the endpoint info so that the Service Configurator also creates this entry for you but in this version, the service configurator is only able to put an entry in handlerMap.config not the app config.

Setup the Endpoint folders

To simplify things, just copy the [endpoint] template folder for your new endpoint and rename it as required. The structure will be as follows.



(1) Setup the 'config' file for the endpoint(s)
   a. The configuration file tells the handler what the incoming operations are and what the responses need to be (the structure of the config file is described in the next section). There is one file for every end-point, for example, Customer.asmx. (In this example, the file would be named **customer**.config)
   b. The config file goes into the endpoints\[**endpointname**]\config folder. For example, endpoints**customer**\config (Only rename the [endpointname] folder, not the others. The folder name does NOT contain the extension (e.g .asmx)
   c. To setup a config file, you can generate an instance from the supplied XSD or, better still, copy the provided example configuration and edit accordingly. (The next section explains the structure of the configuration file)

(2) Setup sample responses
   a. Responses go in the endpoints\[**endpointname**]\responses folder. (No renaming needed here). The response files here correspond to the response names set in the configuration file above. You can generate these from your web service XSD's or use

the instance generation feature of the WSDL Browser UI described in the previous section.

(3) Metadata folder

a. In the current version of the product, there is no explicit runtime need for the WSDL and the schema (although in future versions the system may use the WSDL to infer some behaviour).

b.

Setting up the Endpoint Configuration File

The following screenshot shows a sample HandlerConfiguration for the OOB Handler and a short note about the contents. (This particular screenshot shows the config file for the unit tests and the sample chosen has all possible scenarios and hence the config file looks pretty busy.)

In the datafiles section (either in source or included with the MSI), there is an endpoint named "sample" with the corresponding config file and this has detailed comments on how to set the file up with an example.

A screenshot and brief description is provided here.

```
<HandlerConfiguration xmlns="urn:mockingbird.engine.business.configuration:handlerconfiguration" xmlns:xsi="http://ww
    <Wsdl>sample</Wsdl>
    <ControlSettings>
        <OperationNameInferenceStrategy>RequestRootElementOperationMap</OperationNameInferenceStrategy>
    </ControlSettings>

  <SoapActionOperationMap>
    <Map Key="http://www.contoso.com/:GetPerson" OperationName="GetPerson" />
    <Map Key="Key2" OperationName="OperationName2" />
    <Map Key="Key3" OperationName="OperationName3" />
  </SoapActionOperationMap>
  <RequestRootElementOperationMap>
    <Map Key="GetPersonRequest" OperationName="GetPerson" />
    <Map Key="Key5" OperationName="OperationName5" />
    <Map Key="Key6" OperationName="OperationName6" />
  </RequestRootElementOperationMap>

    <Operations>
        <Operation Name="CreateNewPerson" ResponseStrategy="Default">
            <DefaultResponse ResponseBehaviour="Standard" FileName="CreateNewPersonResponse.xml" ErrorType="None"/>
        </Operation>
    <Operation Name="GetPerson" ResponseStrategy="XPath">
            <XPathBasedResponses>
                <XPathBasedResponse>
                    <Expression>/*[local-name()='Body']/*[local-name()='GetPersonRequest']/*[local-name()='Person']/*
                    <Result>James</Result>>
                    <Response ResponseBehaviour="Standard" FileName="GetPersonJamesResponse.xml" ErrorType="None"/>
                </XPathBasedResponse>
                <XPathBasedResponse>
                    <Expression>/*[local-name()='Body']/*[local-name()='GetPersonRequest']/*[local-name()='Person']/*
                    <Result>John</Result>>
                    <Response ResponseBehaviour="Standard" FileName="GetPersonJohnResponse.xml" ErrorType="None"/>
                </XPathBasedResponse>
        </Operation>
    </Operations>
</HandlerConfiguration>
```

The following are the key elements

- **Control Settings**: Indicates how the system matches up the operation names in the file to the incoming request.
    - An inference strategy of **SoapAction** means that the system looks for an operation name here that matches against the incoming soap action. Please note that this will parse the soap action and use the final token as the operation name. For example a soap Action of http://www.contoso.com/services/customer/:GetCustomer will result in an operation ame of "GetCustomer".
    - An inference strategy of "QueryString" means that the system will look for an operation indicated by a token in the query string. For example a Uri of http://localhost/mockingbird?op=GetCustomer will result in an operation name of "GetCustomer"
    - An inference strategy of "SoapActionResponseMap" means that the system will look for a mapping (in a section below) showing the link between soap actions and operations.
    - An inference strategy of "RequestRootElementOperationMap" means that the system will look for a mapping (in a section below) showing the link between request message root element names and the corresponding operations.

- SoapActionResponseMap: Optional section. Needed only if the Operation name inference strategy matches this. This is useful when the standard parsing rules cannot be applied.

- RequestRootElementOperationMap : optional section. Needed only if the operation name inference strategy matches this. This is useful when modelling services where the soap actions are the same for all operations (this is common in Apache Axis based webservices) and the only hint that we can give the system is the request root element name itself which should correspond to a unique operation.

- **Operations:** There is one Operation corresponding to every Operation/Webmethod in the endpoint/WSDL.

    - **Response Strategy:** The ResponseStrategy tells the system how you want it to locate responses. None – indicates a One Way message, Default –indicates that there is only one response file available and XPath means that there are many possible responses and the right one is selected through applying an XPath expression on the body of the incoming request.
    - **Response Behavior:** ResponseBehavior applies to each response and here you tell the system whether to behave normally or to throw an error (which in turn may be a SoapFault or a Timeout)

Creating/Editing the config file – Tooling

In the current version, the configurator generates a default config file. But you cannot use the configurator to load an existing config file and edit it. We may look into more editing support in future. This is not trivial because MB can be customized so we may end up with separate configuration design time classes that developers need to provide.

To make setting XPath expressions easy, we would recommend using the DanSharp XmlViewer utility.

If you see more GUI support as necessary do let us know (and if you can help with building this, even better ☺)

APPENDICES

IIS 7.x/Win2008 Troubleshooting

HTTP 404 errors when calling ASMX

404 - The resource cannot be found errors when requesting asmx files.

The problem was that the normal HTTP handlers are inherited in the application. You need to specifically remove the handlers for asmx by commenting out the line in the web.config.

Note that for IIS7 the handlers seem to be in the system.webServer section not the httpHandlers one.

Not sure the exact settings that are required but the following seems to work for now:

```
<handlers>
<remove name="WebServiceHandlerFactory-Integrated" />
<remove name="ScriptHandlerFactory" />
<remove name="ScriptHandlerFactoryAppServices" />
<remove name="ScriptResource" />

<add name="ClassicInterceptor" verb="*" path="*.asmx"
type="MockingBird.Simulator.Interceptors.Classic.ClassicInterceptor" />
<!--<add verb="*" path="*.asmx" validate="false"
type="System.Web.Script.Services.ScriptHandlerFactory,
System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35"/>-->
<add name="ScriptHandlerFactory" verb="*" path="*_AppService.axd"
type="System.Web.Script.Services.ScriptHandlerFactory,
System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35" />
<add name="ScriptResource" verb="GET,HEAD" path="ScriptResource.axd"
type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions,
Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
</handlers>
```