Project Mockingbird

Technical Documentation (For v2.0 RTM)

Santosh Benjamin

Technical Documentation

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 March 2010
2.0b	V2 RTM release	Santosh Benjamin	08th Aug 2010

Overview

What is Mockingbird?

Mockingbird is a tool for mocking web services. It is essentially a configurable message interceptor backed by a generic simulation engine that can act as a stand in for any web service.

The three primary usage scenarios for Mockingbird are

- Contract First development
- An integration testing facilitator and
- Isolation of build & dev servers from external dependencies

Usage Scenarios

Imagine you are given the WSDL for a third-party web-service but no functioning system is available yet (it may be a brand new service or perhaps dev/site licenses are being negotiated). You need to get on with development right now. What do you code against? If you are into TDD & MockObjects you may look to mock up an interface corresponding to the WSDL and develop against that. But you may not be using a TDD and mock-objects oriented approach. Even if you do resort to mock objects, that will sort your unit tests out but for the integration testing it may not be possible to target the actual web service (for various reasons) Or it may be that you are maintaining/enhancing an existing system that wasn't coded against interfaces so mocking may not be possible. Or what if you are a BizTalk developer? You cannot mock/inject dependencies into your orchestrations and other components (well, except for pipeline components, but that's another story)!! .

In another scenario, you may be integrating with a third party web service (that may be fronting a LOB application) and setting up a build server and multiple environments (DEV, TEST, UAT etc). But now the vendor says you can only have one license for their web service. Now how do you run DEV, TEST, UAT in parallel with different data sets? Sometimes you may have more than one license, but the service instances may be shared amongst different departments and like all/most business applications, that instance may be subject to operational and maintenance schedules that clash with our build cycles? Your build server (and test process) is then completely exposed to something you don't control.

In general functional and integration testing, there is a limit to how far traditional mock objects can go. They usually stop at the unit test level. It may be possible to use them in functional testing but that is rare. Equally, having the actual web service available is not often possible. Also with a mock service that is primarily config driven, it is possible to provide more exhaustive testing. Setting up mock services manually is a laborious process and there are very few tools that are available to help with this so we developers usually manually code the mock services.

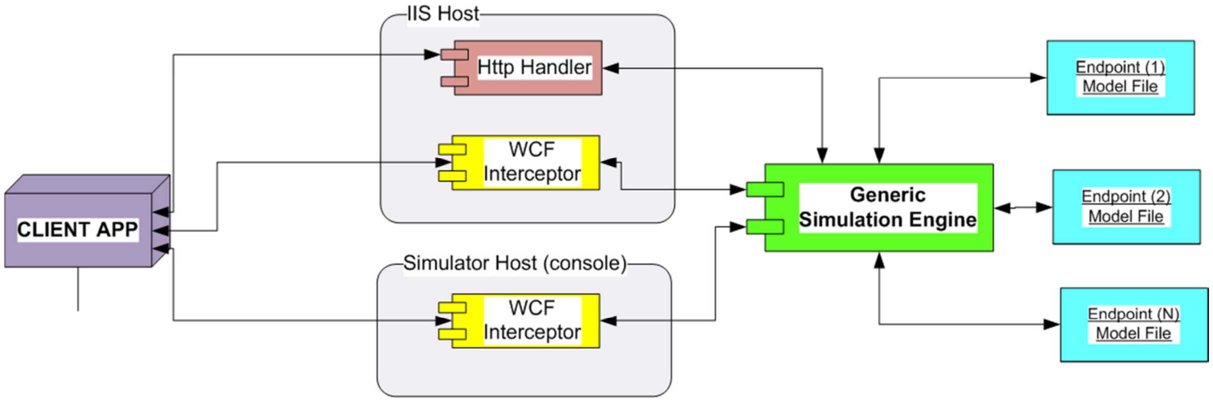
MockingBird aims to make it easy and quick to build configurable mock services. Now there is no need to mess with WSDL.exe and svcutil to generate server side stubs and manually add logic. Simply drop in an XML file representing your service behavior, put in a couple of configuration entries and you're ready to go with a new service endpoint.

Architecture & Implementation

System Architecture

Overview

At a very high level the system looks as shown in the illustration below



The only requirement for the system to work straight out of the box is a “model” file representing the endpoint. The model is currently in XML but there is potential for other representations in future such as XAML.

Components Of MockingBird

There are 3 main components of MockingBird at this time (and many more to come)

- A *Service Simulator* web app which is comprised of - An ASP.NET HttpHandler, A WCF message Interceptor, A WCF SimulatorHost console (for non Http protocol support) and all these are backed up by a generic *Simulation Framework* which is configured to return pre-defined responses using a *model* file for each endpoint that is being simulated.

- A *Service Studio* application that has 3 main modules
    - A ‘Configurator’ module that reads metadata from a given endpoint and automatically generates a fully functional 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.

- A *WSDL Parsing* and *XML Instance Generation* library that is used in the *Service Studio* but can also be used independently.

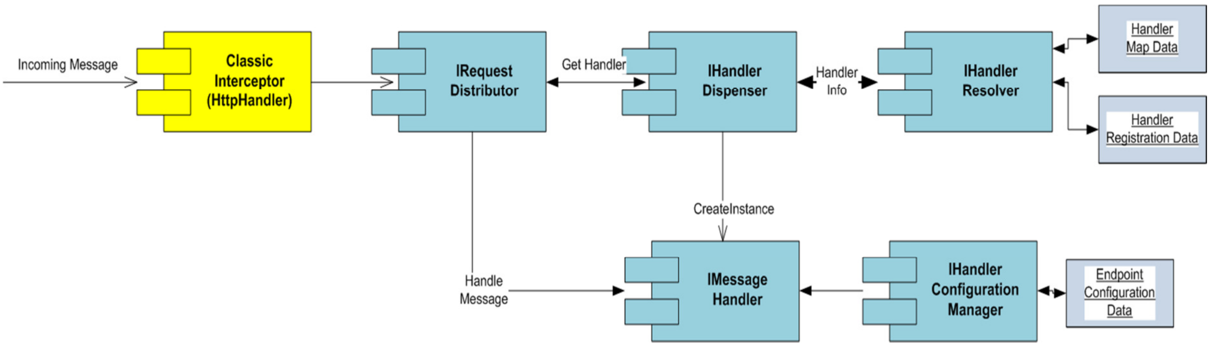
Implementation Overview

The Simulator is essentially composed of two workflows

- Request Distribution: The receipt of a request arriving via any URI and the location of an appropriate handler for the same.
- Response Generation: The process of mapping the request to a model (represented in XML) and executing activities such as request parsing, request validation and response generation.

Handler Resolution Workflow

The illustration below shows the handler resolution workflow for the HttpHandler. The WCF interceptor works in the same way but adds calls a FrameworkFacade that then passes the data on to the IRequestDistributor.



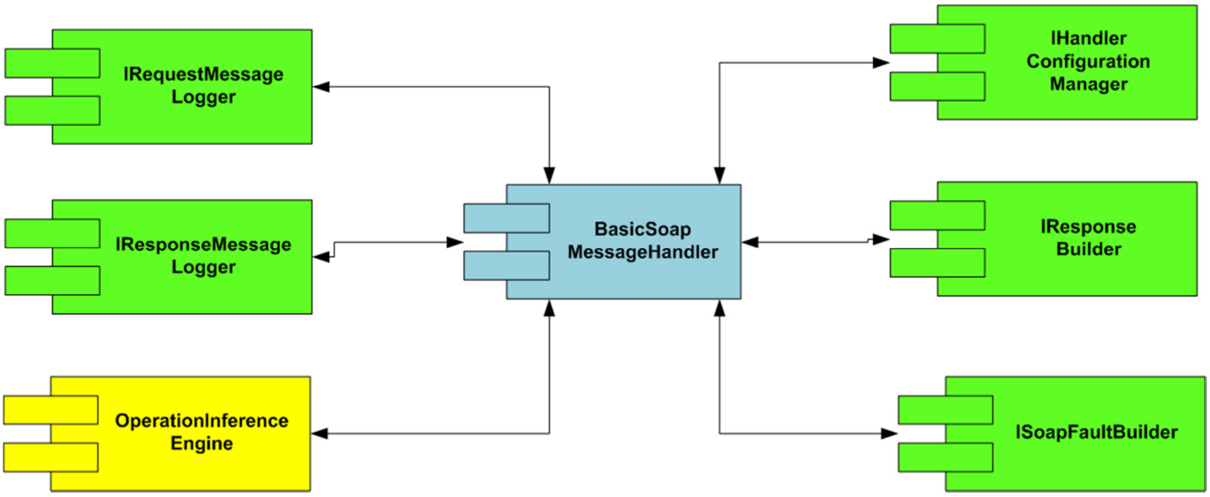
The message comes into the system via the ClassicInterceptor component (a HttpHandler).

This is then passed to the core engine of the application (all components coloured blue). The entry point into the engine is the **IRequestDistributor**. This then gets a **IMessageHandler** instance by calling a **IHandlerDispenser** which in turn uses an **IHandlerResolver** to get the type name of a handler for the endpoint. The Handler Resolver uses two main data structures which are a map of the handler to the endpoint and a registration map of handler name to fully qualified type name. Using this information it returns the name of the required handler to the dispenser. The dispenser then creates an instance of that handler and returns it to the distributor. The distributor then invokes a **handleMessage** operation on the **IMessageHandler**.

(More details on why these classes are required is in a subsequent section below)

Handler Functionality Workflow

The illustration below shows the workflow within the handler and its associates.



The main steps performed by the handler are

- Validation of context and content request format
- Validating and locating configuration information (delegated to IHandlerConfigurationManager)
- Processing of the Soap Request Message
- Logging of the Request Message (delegated to an IRequestMessageLogger)
- Inferring the current operation (delegated to the Operation Inference Engine) which allows for different strategies for operation inference.
- Locating an appropriate response (done with the configuration manager)
- Building the response message (delegated to the IResponseBuilder) or building a SoapFault message (delegated to the ISoapFaultBuilder)
- Logging the response message (delegated to the IResponseMessageLogger)
- Writing the response message back to the Http Context Stream

Implementation Notes & Rationale

- Interception (currently only via ASP.NET) needs to be separate from the core implementation as this is the first component that will change in a WCF version.
- The Request Distributor is a sort of “Workflow Manager” that receives the request from the interceptor and looks for an appropriate handler
- The IHandlerDispenser is a factory for returning handler instances to the distributor. When the HandlerResolver returns a type of handler, the dispenser instantiates it and returns to the caller. The concrete implementation (HandlerDispenser) currently uses a facade over Unity IoC to build up the handler, as the handler itself has multiple dependencies.
- The IHandlerResolver provides an additional layer of indirection. In the current implementation, the concrete implementation is a “ConfigFileBasedHandlerResolver” which reads information from the web.config file.

- All HandlerConfiguration is managed via a IHandlerConfiguration entity. This entity is responsible for interacting with the “model” which governs the handlers execution. The concrete implementation now uses a config file to represent the model, but the idea behind this is that any handler configuration store /model could be used.
- The Handler itself is merely a workflow and all it is expected to do is pull together associated entities such as Request Handlers, Response Builders (including a SoapFault Builder) & the Config Manager to dispatch the response. (Some abstraction is still required – for example, there is no separate RequestHandler class).
- The ‘glue’ class that maintains context is an IExecutionContext which is currently populated from the HttpContext. In future versions this will be loaded from the WCF OperationContext which is protocol independent.

The current handler implementation does have implementation details that haven’t been abstracted into appropriate associate classes and as such, it does more than merely ‘compose’. One of the areas that impact this design is the idea of “extensibility” (discussed below). As the implementation matures, expect this area to change and for functionality to be abstracted into other classes.

Extensibility

General Extensibility via Unity IoC

We make heavy use of the Microsoft Unity Inversion of Control container. The objective is to make the application more dynamic by being able to substitute items such as Configuration Management (perhaps using a database for the source of the configuration info), a handler resolution mechanism (that uses something different from the default name value collection), different response builders etc. Of course, the application is perfectly functional with all the default implementations and so it may never be necessary to change any of this. However, the application codebase benefits from the dependency resolution that Unity provides as well as the increased test surface and coverage (a good side effect of the DI / IoC approach)

Handler Extensibility

In the current version there are two message handlers provided out of the box.

- The *BasicSoapMessageHandler* which ignores SoapHeaders and returns responses based on XPath and other configuration parameters.
- The SoapMessageHandler is similar to the “basic” version. It looks for specific soap headers and then uses the values of the xpath queries run on the header elements to build headers for the corresponding responses. At the current time, this is a specific implementation for a use case in the development team and in the next version it will be made more generic.

The framework is designed to be extensible. One of the main extension points is in the area of handlers. So, if you need to write a handler to process custom headers or return responses in a different way, you should be able to write a handler and plug it in via configuration. For instance, if you are dealing with WSE2 and WSE3, then the Soap Headers are a key area of work. The MessageHandlerBase class also does most of the heavy lifting so you only need to override a few methods. Take a look at the SoapMessageHandler and BasicSoapMessageHandler to see how the current implementation uses and overrides the base class.

Extensibility & Implementation Roadmap

While writing a custom handler is definitely possible now, in the long term I may change the approach to extensibility. Ultimately the tool processes a soap request and returns a soap response. It may need to act on a custom soap header or just act on the body and return a response. Being able to plug in “header actions” or “body actions” could be more useful than attempting to plug in an entirely new handler as there really isn’t that much variation in message handling anyway.

If we take the ‘action’ approach, then parsing the header or body using XPath is only one action. We could think of other actions that could execute on the soap header or body with any required logic in them. So, the config file would now change to ask for things such as “ActionType” (Header / Body”), ActionName (XPath / other) and if the Action Result is , say, true, then the associated response file would be returned. Response Building could also be extended to add custom soap headers etc.

Another analogy that has been suggested for a future version is similar to BizTalk pipelines. For every URL we can provide a ‘virtual’ handler which in effect is just a sequence of components all implementing the same interface. This will improve composability and reduce the code surface as we wont need to implement new handlers anymore, just compose from existing parts or add well-defined new parts.

With Windows Server AppFabric now available and WF4 providing a simpler and more robust model for flow based tooling, MockingBird could make use of both these technologies. Using AppFabric would mean we don’t need to care about the separate console host and a single AppFabric host will serve all protocols. The WCF4 Routing Service could replace the current interceptor.

Considering that MockingBird is really about workflows (handler location and response construction, the component parts of the framework are great candidates to be WF4 activities which means that any new handlers will now be WF4 workflows themselves. WF4 will also allow us to take even more advantage of AppFabric management and instrumentation which makes our mock services much more manageable.

Ultimately v2 aims to be a good working foundation for something that will grow to be highly modularised and extensible and so at this point, it is not fully there yet. The plugin model may utilize MEF or use Unity as a basis.

Key Concepts

Services and End-points

(1) A SOA business service may have many endpoints. We care only about mocking the endpoint not about the SERVICE. (ie: the service itself does not enter into the design /datafile structure at this time).

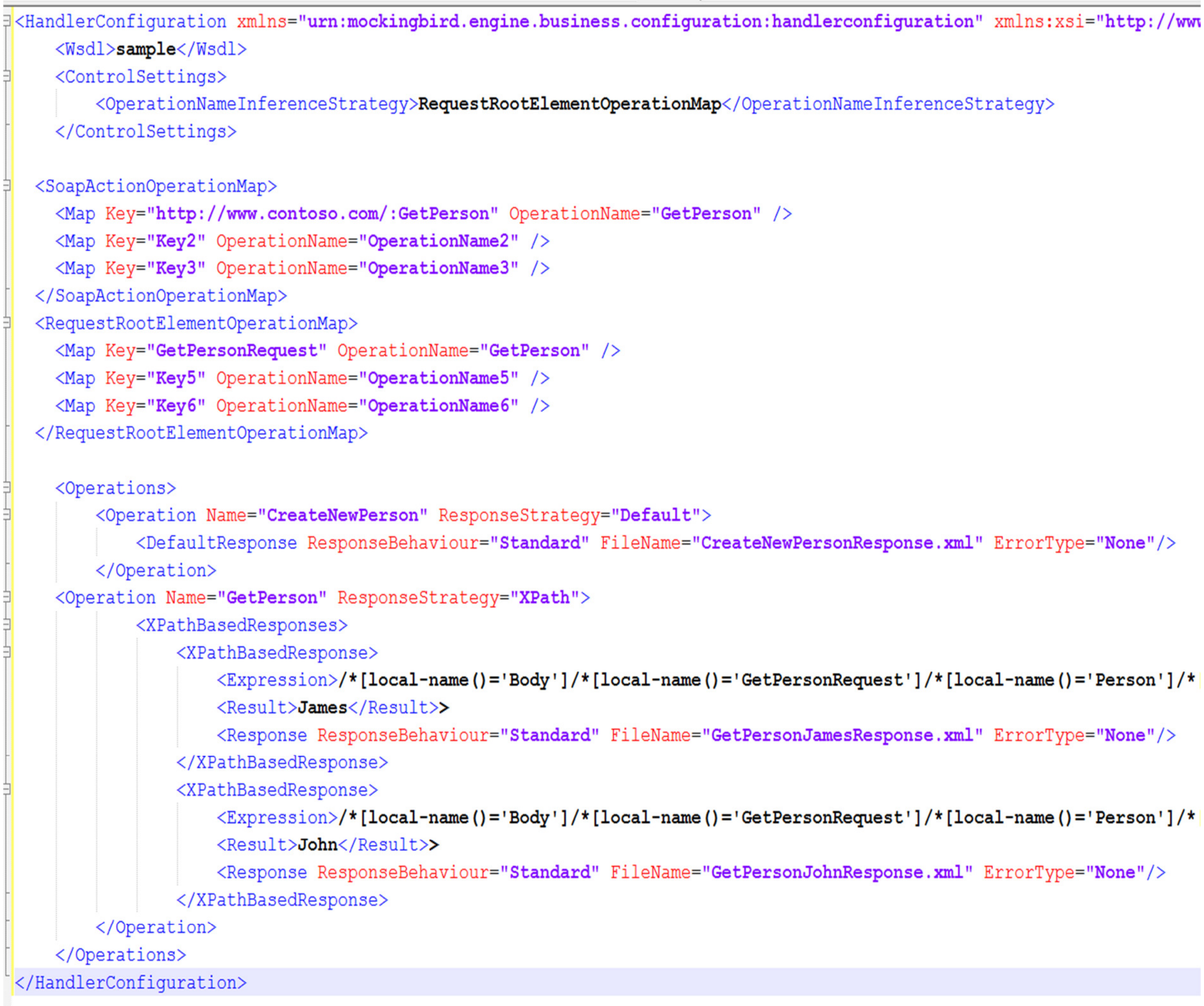
(2)An endpoint may have one or more operations. There is usually one 'contract' for this endpoint and it usually expressed in a single WSDL. The endpoint will, in turn, be 'supported' by a config file and the WSDL may or may not be used when processing the config file (or it may be used for some operations and not for others).

(3)The entire endpoint conforms to one protocol. For instance, if WSE is used, then all operations will be controlled by the WSE SoapHeaders. We dont mix and match protocols in endpoints (we do not go into 'bindings' in v1). Therefore we have a single protocol Handler associated with the endpoint and ALL the operations/messages in that endpoint. This means that we define every URL that MockingBird is going to support and we associate it with a handler

The Endpoint Description Model

As noted above, the heart of the system from a user point of view is the model describing the endpoint. This is in XML now.

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)

Implementation Highlights

Framework Configuration

This section explains how the various components make use of configuration info at runtime.

The key aspect about configuration is that we don't rely on storing info relating to the MockingBird system in web.config. It is externalized into its own config file. Similarly entlib logging and log4net logging have their own config files. In the case of the entlib logger we have completely overridden the configuration so that the web.config and app.config files do not need any entlib related entries. In future versions the WCF endpoints will also be externalized (AppFabric may make this easier).

Important Infrastructure

The MockingBird.Common.Infrastructure library contains a number of components used by the Simulator and other solutions. The key things to mention here are

- Log4net is in a separate config file and surfaced through a log4net logger implementing the ILogger interface.
- EntLib logging is in a separate config file and surfaced through an EntLib logger implementing the ILogger interface
- Unity Config is in a separate file and surfaced through a DependencyResolver class.

3rd Party Libraries

The primary 3rd party libraries are

- Log4net
- EntLib logging
- Unity IoC
- SubText HttpSimulator (used in functional testing)
- MoQ (for unit testing)
- Gallio and MbUnit (unit test framework)

The default logger configured in the Unity config is the Log4net logger. However this can be swapped out for the EntLib logger if required.

Detailed Scenarios for using MockingBird

Contract-First Development

In Contract-First development, the specification for the service contract/interface is first developed before the actual implementation. There are two approaches for Contract-First namely Code-First (where you write the interface in C#/VB.NET and use that code to generate other artifacts such as WSDL or other metadata endpoints) and WSDL First (where you first write the XSDs for the messages and compose the WSDLs and then use that to generate the .NET code.

In the current version Mockingbird requires a WSDL but does not prescribe whether the WSDL should have been created first or the .NET code. The GUI parses a given WSDL and sets up the service so as long as the WSDL is valid and all the schemas can be discovered (embedded or linked) there should be no problem. The engine does not depend on the WSDL (although the WSDL location is specified as an optional element in the configuration schema).

There are two use-cases within the Contract-First development scenario.

- As a web-service consumer
- As a web service provider

As a web-service consumer

You may be integrating with a web-service that is external to the immediate system under development. This may be a third party web service (installed on or off-premises) or a web-service interface to another subsystem that a part of your team is building/has just built. Often when integrating with third-parties, we are first provided with WSDLs and Schemas and eventually we get access to a functioning system.

In this scenario, Mockingbird gives you complete control over the requests that you send to it and the responses you get back. This means that you can concentrate on making your calling application work and handle responses correctly, without worrying about unpredictable behaviour from the external web-service. As such it is an excellent unit/integration testing tool. Because the message responses are simply Xml files on your local PC you eliminate the following difficulties:

- Someone may be developing the web-service and taken it off line
- Or messed it up so it throws an exception
- The underlying database might be down
- You or your web services might not have the right permissions to call it
    - You might not be able to call it twice with the same data (say if you're testing your *delete* functionality).
- Licensing Issues: The vendor may allow you to install only one instance of the web-service which makes it very difficult, if not impossible, to run parallel environments like DEV, TEST, UAT with their own datasets & versions.

As a web-service provider

You may be providing a new web-service to an external partner or another feature team. In these situations it is helpful to be able to work out the specifications for the message and data contracts first and then embark on development (with the appropriate processes in place for change-control etc).

In this scenario, you can rapidly provide a web-service interface for the consumers so that your development schedules are not too dependent on each other and if the contracts change as you develop, updated webservice interfaces can be quickly provided without the hassle of providing a complete install and all the underlying infrastructure that goes with it.

Additionally the other side of the consumers points listed above also apply to you as a provider.

What about testing against the actual system?

Of course you do eventually need to perform functional and integration tests against the actual external system because it is not possible to simulate all the exceptions and data conditions of a line-of-business application in a mock service. Mockingbird just helps you get a working interface with a comprehensive base of response messages that will help in getting development off the ground

As an Environment Isolator

Just as we need predictable responses for unit tests, it is also beneficial if our build / dev servers don’t have to rely on external web services being available. instead of depending on an external system, we host Mockingbird on our local webserver. We can then change the responses it sends back to suit our purposes, and define simple business rules that determine which of several messages to return to us. Network problems are eliminated, you are not at the mercy of external changes, and your tests are endlessly repeatable. Because it is hosted locally, your tests should also work on any other machine that downloads has all the config files to support Mockingbird.

Acknowledgements

Origins of the tool

Mockingbird started life as ‘MockWeb’ an internal tool that my former colleagues (Senthil Sai and Will Struthers) and I developed. All credit must go to them as Senthil first came up with the concept and Will then contributed a lot to the code-base. It started life when we got rather fed up with having to set up multiple instances of a third party service and build data-sets just to help with testing our BizTalk orchestrations reliably. It grew rapidly and organically. While we felt that this concept and tool would be useful to the .NET dev community at large (*not just BizTalk teams*), the structure of the codebase (at the time) would not lend itself to easy extension and needed to be refactored.

I have since completely rewritten MockWeb and hence the new name.

Contributors & Supporters

I would also like to thank Nick Heppleston for his early adoption and feedback which helped build more quality into the tool. I would also like to thank Yossi Dahan and Michael Stephenson for their support.