Synapse Guide draft

Apache Synapse is a mediation framework for Web Services. Synapse allows messages flowing through, into, or out of an organization to be mediated, including aspects such as:

- Logging, service lookup, performance mediation
- Versioning, failover, monitoring
- Fault management, tracing

Getting started

Although there is a cleanly defined division between Synapse and Axis2, the Synapse system relies on Axis2 to run. Firstly, it uses the AXIOM object model, and secondly it uses Axis2 as a listener and sender for Web service requests.

There are two ways to set up the Synapse server.

1. synapse.war which can be deployed in a servlet container.
2. Lightweight server which can be run under Axis2’s SimpleHTTPServer (a simple lightweight HTTP server option that does not require a Servlet Engine)

You can either download these or build them using Maven.

You can build the war file by using the command:

```
maven dist-bin
```

which creates both the WAR and binary distribution JARs.

If you to use synapse.war, deploy it in into your favorite servlet container.

Once it's exploded, you will see in WEB-INF the *axis2.xml* which has been configured to execute Synapse properly and *synapse.xml*, which will hold the rules pertaining to messages passing through Synapse.

If you wish to use the standalone server, unzip the Synapse-M1-SNAPSHOT.zip. In the bin directory you will find a script called:

```
synapse-lightweight [.sh or .bat]
```

You should also see a directory called synapse-repository. In there you should find the axis2.xml and synapse.xml config files. The axis2.xml should not need to be modified.

The command line for synapse-lightweight takes the repository directory and listening port, so:

- `sh bin/synapse-lightweight.sh synapse-repository 8080 [Linux]`
- `bin\synapse-lightweight synapse-repository 8080 [Win]`

Processing model
Synapse has an overall model under which there are two ways to extend the framework.

1. Using the SPI, developers can build *Synapse Extensions*, which extend both the functionality and the XML configuration syntax of Synapse.
2. Using the API, developers can build *Mediators*, which extend the functionality of Synapse but use the existing XML syntax.
3. There are also built-in mediators that do common tasks like logging, redirection etc.

Typically users of Synapse extend the function using mediators, while the Synapse development team can extend the core by building extensions.

A synapse deployment attaches to one or more transport listeners, and mediates messages from those listeners. One of the key decisions is how to “attach” mediators to messages.

Rules
By default Synapse will execute all defined mediators against a given message, but this can be affected by using simple *rules.* Synapse has two predefined rules: <xpath> and <regex>. *xpath* evaluates and XPath expression against the message, while *regex* matches a regular expression against one of the message headers (such as the wsa:To address).

Synapse also has two simple rules <in> and <out> which process only request or response messages (as seen by the target service).

Stages
As a message goes through the Synapse engine, it can pass through multiple stages. Each stage is a way of grouping and organizing mediators and rules. A stage simply gives the group a name.

An example
At this point an example would be useful.

```
<stage name="stage1-all">
     <!--This enables the addressing module
          which looks at wsa headers -->
     <addressing/>

     <!–Logs the message -->
     <log/>
```

```
</stage>

<stage name="stage2-service-specific" >
     <regex message-address="to"
               pattern="http://xmethods.*">
     <header type="to"
                value="http://64.124.140.30:9090/soap"/>
          </regex>
</stage>

<stage name="stage3-send-all">
     <send/>
</stage>
```

This example demonstrates *stage*, *regex* and some built in mediators: *log*, *addressing* and *header*. It does not demonstrate the *xpath*, *in* or *out* rules.

Every stage will be executed for each message. The first stage does initial processing including parsing the addressing headers and logging the message.

The next stage is using a regex rule to redirect every message addresses to xmethods.com and xmethods.net to the real SOAP address of the XMethods quote service.

Finally the last stage sends the message on. For responses, the messages come back through the same stages. This time the message will not be redirected because the “to” address on the response will not match xmethods.

In and Out
We could have been more explicit that the redirection is only designed to apply to “in” messages by using the <in> rule.

```
<stage name="stage1-all">
     ...
</stage>

<in name="stage2-service-specific" >
     <regex message-address="to"
               pattern="http://xmethods.*">
     ...
</in>

<stage name="stage3-send-all">
     ...
</stage>
```

There is a corresponding <*out*> rule.

References
In order to make the configuration more re-usable, every rule, stage or mediator can be named:

```
     <stage name=”thisname”>
```

The name can then be used to “refer” to the mediator.

So
```
     <ref ref=”thisname”/>
```

will cause the same processing to happen as if the stage had been included at that point.

For example:
```
<in>
     <stage name=”both”>
     . . .
     </stage>
     <stage name=”inonly”> …</stage>
</in>
<out>
     <ref ref=”both”/>
</out>
```

[Please note this is one area where we expect to do considerable work ☺]

Never

This is a stage where none of the children get executed. Its purpose is to allow you to place rules and mediations and have them not executed but instead refer to them from one or more other places.

So the following may be deemed equivalent to the previous example

```
<in>
     <ref ref=”both”/>
     <stage name=”inonly”> …</stage>
</in>
<out>
     <ref ref=”both”/>
<out>
<never>
     <stage name=”both”>…</stage>
</never>
```

Content based routing
We can further improve this example by adding some “content-based” routing. Using an <xpath> rule we can make tests within the XML. For example, we could decide not to allow stock ticker queries against certain companies whose share prices we were jealous of – MSFT say :-).

To do this we can add a rule:

```
<xpath expr="//*[Symbol='MSFT']">
     <fault/>
</xpath>
```

This rule identifies any messages with a tag “Symbol” whose content is MSFT. The <*fault*> mediator returns a fault to the client.

We can place this rule under the regex rule, so it only applies to requests aimed at xmethods.*:

```
<regex message-address="to" pattern="http://xmethods.*">
     <header   type="to"
               value="http://64.124.140.30:9090/soap"/>
     <xpath expr="//*[Symbol='MSFT']">
          <fault/>
     </xpath>
</regex>
```

Note that the rules, like the stages, can have more than one child. While it isn’t fixed in Synapse, the built-in rules and mediators all use the same “plan” to execute their children, which involves executing in the lexical order that they occur in the synapse.xml.

Samples

Logging

The system ships with a couple of samples. These include sample clients and appropriate synapse.xml intermediary configurations.

The first sample demonstrates the logging facility. Here is a simple synapse.xml:

```
<synapse xmlns="http://ws.apache.org/ns/synapse">

     <addressing/>

     <log/>

     <send/>

</synapse>
```

The logging uses the Log4J/Commons Logging support in Apache. You can configure it using log4j.properties.

The sample client is a standard Axis2 client built to run against the XMethods Quote Service. However, it has been modified to use a different transport address from the Web Services Addressing TO header. In other words, the SOAP envelope is addressed to the XMethods service, but the actual HTTP request goes to Synapse.

The sample client has three (optional) parameters:

```
StockQuoteClient SYMBOL XmethodsURL TransportURL
```

e.g.

```
     StockQuoteClient IBM http://64.124.140.30:9090/soap \

          http://localhost:8080
```

The sample synapse.xml can be used to demonstrate a few simple behaviours.

1) Firstly try this:

```
     StockQuoteClient IBM http://64.124.140.30:9090/soap \

          http://64.124.140.30:9090/soap
```

This will bypass Synapse and simply call XMethods.

2) Now start Synapse and try

```
     StockQuoteClient
```

on its own. You should see the messages being logged as they pass through Synapse.

3) This time try

```
     StockQuoteClient IBM urn:xmethods-delayed-quotes
```

This should hit a regex rule which replaces the “virtual URI” that is in the wsa:To header with the real URL.

4) Now try StockQuoteClient MSFT which should hit a “content-based” xpath rule.