SOA governance using WebSphere DataPower and WebSphere Service Registry and Repository, Part 2: Authoring and enforcing custom policy vocabularies

Thomas Burke (burket@us.ibm.com)
Advisory Software Engineer
IBM

Mario E. De Armas (drhoek@us.ibm.com)
Senior Technical Staff Member
IBM

Evan Jardine-Skinner (eskinner@uk.ibm.com)
Software Engineer
IBM

Oswaldo Gago (gago@us.ibm.com)
Advisory Software Engineer
IBM

Part 1 of this article series introduced the new message mediation capability that WebSphere® DataPower implemented in the 5.0.0 version of the appliance firmware. This article, Part 2, will explain how to extend these capabilities even further by enabling the use of custom policy vocabularies to deploy specific proxy processing patterns not covered by built-in policy domains.

View more content in this series

Introduction

Part 1 of this article series introduced the new message mediation capability, which WebSphere DataPower implemented in the 5.0.0 version of the appliance firmware. This article, which is Part 2 of the series, will explain how to extend these capabilities even further by enabling the use of custom policy vocabularies to deploy specific proxy processing patterns not covered by the built-in policy domains. The article will guide you through the steps necessary to define a customized policy vocabulary and enforce it as an SLD or SLA using WebSphere DataPower and WebSphere Service Registry and Repository (WSRR).
This article is written with the assumption that you are familiar with XSLT, XML, WS-Policy, and DataPower configuration. You can learn more about WS-Policy by reading the WS-Policy Resources included at the end of this article. You can learn more about DataPower configuration and DataPower processing policy from DataPower SOA Appliance Handbook or the DataPower Information Center.

Downloads

This article provides the following files for you to download:

- **yourco.com-example.xsl**: This is the sample custom policy presented in this article.
- **mediatemessage-example.xml**: This is the sample policy file that demonstrates the use of MediateMessage assertion.
- **datapower-export-configutils.zip**: This is the DataPower config export, which contains utilities to process the export.xml to prepare it or inclusion in a policy mapping stylesheet, and a DataPower configuration schema validation.
- **YourCoExampleWidget.zip**: This is the WSRR widget sample that implements the YourCo.com custom policy authoring.
- **YourCoExampleWidgetProjects.zip**: This is the source code for the WSRR widget sample.

Benefits of a custom policy vocabulary

WebSphere DataPower Appliance can enforce out-of-the-box WS-SecurityPolicy and WS-MediationPolicy vocabularies. These policy vocabularies are useful, but the possibility exists that these vocabularies, built to provide general purpose web service solutions, will not completely meet your specific business needs. For example, the ability to define your own vocabulary becomes critical when the need to guarantee maximum reuse with least possible risk will force the policy semantics to be elevated to a much higher level, thus mitigating many of the risks introduced by the flexibility of finer-level assertions in those policy vocabularies. In other words, the level-of-detail (LoD) of a custom policy vocabulary is dependent on each enterprise’s need to balance the policy's agility factor (how quickly it can react to changes in business needs) to its operational risk and cost factors (how much does its flexibility cost in terms of testing and operational stability risks).

Therefore, the benefits of creating customized policy vocabularies can be great (in consumability by the IT management process and mitigating operational risk and costs) to most enterprises who want to evolve beyond the general-use policy vocabularies provided by DataPower.

A way to illustrate this is to assume that your enterprise has defined the following message handling requirements for the services they provide:

- **All services must enforce at least one of following levels of protection**:
  - **Standard**: This level of protection requires only username/password authentication via an HTTP or `<soap:header>` authentication header.
  - **Secure**: This level of protection requires that all requests contain a WS-Security or LTPA token for authentication. All messages must be signed and encrypted. This requires audit logging.
• **Restricted**: This is the same security requirement as the "Secure" level above, but it adds that authorization to access specific service operation or resource must be authorized using the enterprise access-management enforcement point.

• **All messages must be validated or transformed according to following guidelines:**
  • All requests that adhere to the V1.0 schema must be rejected since it has been retired and is no longer in service (supported).
  • All requests that adhere to the V1.1 and V1.2 schemas must be converted to the V2.0 schema. Responses arrive according to the V2.0 schema and must be converted to adhere to the schema of the original request. This provides backward compatibility to older versions of the data schema and allows for a managed deprecation and migration process.
  • All messages that adhere to the V2.0 schema require no additional processing on the request and response flows. This is the latest version of the data schema that the backend service currently supports.

• **All messages must be routed to one of the following defined backend types for processing:**
  • **Standard**: There are no special routing requirements. Use the default backend specified by the WSDL service.
  • **High availability**: This routes to a 24x7 highly available backend.
  • **High availability with low latency**: This routes to a 24x7 highly available backend that has been optimized for a low latency response.

The usual way to enforce these business requirements has been to configure processing actions for each service proxied by DataPower, according to the requirements defined for that service. This requires configuration changes to the processing rules for each service, which means that each service deployed requires unique development, unit test, and pre-production Quality Assurance (QA) efforts. All of this additional cost is multiplied for additional non-functional requirements that SLAs must support, such as Quality of Service (QoS) agreements.

In contrast, enforcing these business requirements as policy assertions (exposing limited variability) provides the benefits of reduced development costs (define, author, and test assertions once, then apply them many times as needed). This also reduces runtime stability concerns because the configuration is auto-generated by the appliance in a repeatable and deterministic process, without the possibility of introducing human error or unexpected variability.

Furthermore, the ability to re-use policy assertions with different SLAs (such as multiple service consumers and providers) greatly multiplies the benefits of adopting policy enforcement. As an example, applying the policy vocabulary as described above with the SLA enforcement capability delivered in DataPower firmware version 5.0.0 allows the policy to route "Gold" customers to a "High Availability" backend and all other (non-"Gold") customers to a "Standard" backend.

### Policy enforcement on a DataPower appliance

Now that we covered the benefits of using a custom policy domain vocabulary, you are probably ready to dive into an example. But before we do that, let's go over some background information.
to help you understand the role that this example will play in provisioning service policy and how it impacts the operation of the appliance.

The W3C standard called "WS-Policy" provides a framework for policy solutions to describe and consume business operational requirements using a specific data model that helps capture the business intentions in policy form. The WS-Policy - Framework specification dictates that policy expressions be represented as XML. The DataPower appliance, acting as a Policy Enforcement Point (PEP) consumes WS-Policy expressions and converts them into native (enforceable) configuration. Part of the policy-to-configuration processing is handled by the DataPower Policy Framework subsystem.

This DataPower framework generates an effective policy by normalizing policy expressions into alternatives, and alternatives into assertions. The conversion of individual policy assertions into DataPower configuration is delegated to the DataPower Policy Mapper stylesheets, which consume WS-Policy XML and create DataPower XML config.

The DataPower framework then assembles the responses from the individual DataPower Policy Mapper stylesheets into the final configuration to be applied to the service gateway affected by the policy. This "final assembly" step ensures that the configuration is generated correctly, even if more than one policy vocabulary is used in an alternative. Figure 1 provides a visual representation of this process flow and components.

Figure 1. DataPower Policy Framework overview

The DataPower Policy Framework component is an extensible policy processor that can generate DataPower configuration from WS-Policy assertions. The appliance firmware release 5.0.0 includes, out-of-the-box, several policy domains that you can use immediately to affect gateway enforcement behavior. It also provides extensibility to add custom policy domains to the Policy Mapper.

To add new custom policy domains, a Policy Mapper stylesheet must be supplied for each new policy domain that you want to enforce with the PEP. Creating such a conversion mechanism
requires that you understand how to use DataPower processing rules, processing actions, and any prerequisite objects (like SLM and AAA Policy) to achieve your policy enforcement goals. For example, ask yourself: "How would I configure a processing rule directly in a Web Service Proxy to achieve the desired results?". As a matter of fact, it is a good idea to actually create the initial implementation on a processing rule in the WebGUI, test it, and export the processing rule to use as the starting point for your policy mapping stylesheet. For more information about DataPower processing rules and processing actions, consult the DataPower Information Center and DataPower SOA Appliance Handbook.

Defining a custom policy vocabulary

A little planning goes a long way when it comes to defining a custom policy vocabulary. Once you understand the business requirements, there are a few simple planning steps that will directly apply to your implementation. Careful planning and realistic goals are important to avoid unnecessary complexity later on.

The first step is to pick an XML policy namespace for your assertions. The namespace is important because it eliminates the possibility that the assertions that you define will collide with assertions created in another policy domain. We will use http://yourco.com/policy/standard/2012-08 for this article. The namespace prefix for this namespace will be ycps. Note that a date was included in the namespace. It is a best practice to include a date or version identifier in the namespace to support the notion of future versions of the policy domain.

The next step is to define the assertions that you want to enforce. In the example at the beginning of article, we defined three top-level business requirements to enforce in policy. We recommend that policy vocabularies closely resemble the business IT semantics being applied, as doing so improves users' ease of adoption and auditing of the policy governance platform (such as WSRR). Therefore, we will map the original three business requirements listed above to individual assertions so that our new custom policy vocabulary contains three assertions.

A sample syntax for these assertions is found below in Listing 1. The assertions mentioned in this article are for discussion only, and provide no function in the template provided. Even though we show how to insert a DataPower configuration into the policy assertion stylesheet, implementing these assertions should be considered an exercise for the reader.

Listing 1. Define the syntax of each policy assertion

```xml
<ycps:ProtectService>
</ycps:ProtectService>

<ycps:MediateMessage/>

<ycps:RouteMessage>
</ycps:RouteMessage>
```

As you can see from the three assertions defined (ProtectService, MediateMessage, and RouteMessage), no information about the specific behavior or how it will be enforced is provided in the assertion. It is not intended to show what specific actions the PEP takes to handle a restricted
message, or where the high availability servers are. This omission is intentional, because doing so decouples the assertion (business requirement intent) from the implementation on the PEP, and permits the interpretation of "secure" to change when business IT security requirements change by altering the implementation (such as mapping) on the PEP. Note that care should be taken when making changes to assertion behavior. Consider changing the namespace of the policy domain for drastic changes in behavior.

Note that the order of assertions within a policy is important. For example, cryptographic operations (that "Secure" and "Restricted" protection levels would require) would need to occur before message payload mediation is applied during transactions request processing.

**Policy Parameters: The finishing touch**

Part of a good design for a policy assertion is knowing what should be expressed as an assertion parameter and what should be expressed as part of the PEP implementation. Occasionally, you run into a situation where neither case fits. The RouteMessage assertion in WS-MediationPolicy is a good example. WS-MediationPolicy uses the RouteMessage assertion to route messages to an alternative endpoint. But, what if the destination requires SSL? On a DataPower appliance, the SSL handshake requires an SSL Profile. If you used a "route" processing action in this capacity, you have to specify an SSL Profile object name as a part of the configuration for the route action. That is acceptable for a single processing action, but how should this be done in a WS-Policy assertion?

The problem is that this SSL profile object name cannot be specified as a parameter for the policy assertion (such as in the policy assertion) because that ties the assertion to a specific PEP implementation (WS-Policy assertions should be PEP-agnostic). In the same vein, it also cannot be coded into the implementation of the assertion. To do so greatly limits the applicability of the assertion. The assertion only works for one backend, or worse, the assertion requires an SSL Profile with a particular name. The best solution for this conundrum is to connect the dots using a policy parameter to tie the behavior specified in the assertion to the actual implementation. That way, the policy remains PEP agnostic and the assertion remains flexible.

**Introducing the custom policy domain example**

Now that you know that the Policy XML is converted to a configuration XML with a policy mapping stylesheet, let's take a look at the example that is provided with this article.

The first thing to notice is the declaration of the policy domain namespace for the appliance, as shown in Listing 2. This is the part that tells the appliance which XSL transform to use for policy assertions that are defined in the http://yourco.com/policy/standard/2012-08 policy namespace to convert them to a configuration.

**Listing 2. Declaring a policy domain in a policy mapping stylesheet**

```xml
<dpe:summary xmlns="""">  <!-- Comment the following line to prevent this policy domain from being processed --
    <operation>xform</operation>
    <description>YourCo.com Custom Policy Support</description>
</dpe:summary>
```
The next thing to notice is the stylesheet entry point `<xsl:template match="/">`. This template takes care of a couple of housekeeping matters (such as figuring out the name of the input and output contexts - more on that later), then it applies the templates for each assertion it finds in the policy, as shown in Listing 3.

**Listing 3. Applying templates for each assertion**

```xml
<!-- process each assertion in the alternative -->
<xsl:template match="wsp:All">
  <xsl:apply-templates mode="assertion"/>
</xsl:template>
```

Note that a single WS-Policy expression may contain multiple policy alternatives, so the policy mapping framework on the appliance handles policy translation in such a way that the policy mapping stylesheet is called separately for each alternative encountered in policy. This means that you do not have to be concerned about how to provide configuration information separately for each alternative or the order that the alternatives are processed.

Each alternative may contain more than one policy assertion ("1..*" in fact), so you must ensure that each assertion that you process is handled properly by wrapping the configuration for each assertion in a `<dppolicy:config>` element as shown in Listing 4, and setting the `uri`, `assertionNo`, and `direction` attributes.

- The `uri` attribute is used to tie the configuration you generate to your policy domain. The value is set to the namespace of your policy domain.
- The `assertionNo` attribute is used to indicate the relative order that the assertion appeared in the alternative. This is necessary because the policy mapping framework of the appliance attempts to enforce the assertions in the order that they have been normalized into the final effective policy. Note that the order of enforcement is not guaranteed, and the WS-Policy Framework specification does not require that assertions be enforced in the order that they appear in a policy alternative. See [Web Services Policy 5.1 Framework: Policy Alternative](#).
- The `direction` attribute dictates the processing direction that applies for the assertion (for example, request, response, or both). Valid values for the direction are `request-rule` and `response-rule`. If you omit the direction attribute, then the actions supplied apply to both rule directions.

The contents of the `<dppolicy:config>` element look familiar if you have inspected the configuration of a DataPower appliance. The code shown in Listing 4 shows how the template to create the configuration XML for the MediateMessage assertion.

**Listing 4. Generating configuration for a single assertion**

```xml
<!-- TEMPLATE -----> <!-- assertion "MediateMessage" --> <!-- This basically "does nothing" by performing an identity transform to copy the input to the output. -->
<xsl:template mode="assertion" match="ycps:MediateMessage"/>
```
<!-- This assertion contains examples of best practices - Using the correct input context ('input-context') and output context ('output-context'). - Use policy key to correlate processing actions to policy. - A method for creating unique names for StylePolicyAction (see: "actionName") It also contains some tips like - Using xsl:message with dpe:priority="error" for debug purposes. - The practice of placing the generated configuration XML into a variable "config" so it can be sent to the log, or written to the temporary directory using the dump-nodes extension function. -->

<xsl:message dpe:priority="error">Enter template MediateMessage: . . .</xsl:message>

<!-- The framework passed in a policykey value, this must be added to the processing actions in order to properly instrument source policy origin information into monitoring subsystem and UI presentation. -->

<xsl:variable name="policyKey" select="./@policy-key" />

<xsl:variable name="config">

<!-- generate processing actions (assertionNo = position of assertion in the normalized policy alternative) -->

<dppolicy:config uri="{$nsuri}" assertionNo="{position()}">
  <xsl:variable name="actionName" select="concat($header/dppolicy:PolicyID,'-xform-',position(),'-Mediate')" />

  <!-- Create the name of the action. It must be unique within the DataPower Application Domain. Notes: The PolicyID will help you make the name unique, because it is unique per attachment point. It is a best practice to identify the type of action in the name, to simplify debugging. It is also a best practice to provide a hint of the assertion this action supports in the name -->

  <xsl:variable name="actionName" select="concat($header/dppolicy:PolicyID,'-xform-',position(),'-Mediate')" />

  <!-- Create the action using the name created above -->

  <xsl:element name="StylePolicyAction">
    <xsl:attribute name="name">
      <xsl:value-of select="$actionName" />
    </xsl:attribute>
    <xsl:attribute name="Type">
      <xsl:text>xform</xsl:text>
    </xsl:attribute>
    <xsl:element name="PolicyKey">
      <xsl:value-of select="$policyKey" />
    </xsl:element>

    <!-- The input and output contexts usually require special consideration because you may want the result of one assertion to serve as the input of the next assertion, in which case the Input should not be INPUT and the Output should not be OUTPUT. Instead use the contexts specified in the notepad (which have already been placed in the local variables shown. -->

    <xsl:element name="Input">
      <xsl:value-of select="dpe:local-variable('input-context')" />
    </xsl:element>
    <xsl:element name="Output">
      <xsl:value-of select="dpe:local-variable('output-context')" />
    </xsl:element>
    <xsl:element name="Transform">
      <xsl:text>store:///identity.xsl</xsl:text>
    </xsl:element>
    <xsl:element name="OutputType"><xsl:text>default</xsl:text></xsl:element>
  </xsl:element>
</dppolicy:config>

</xsl:variable>

<!-- Send the config to the console, or whatever portion of it will fit (roughly 3k) -->

<xsl:message dpe:priority="error">Exit template MediateMessage:
  <xsl:copy-of select="$config" />
</xsl:message>
Using the policy editor to create the XML for an assertion

This section shows how to author a policy assertion template using the WebGUI to create a processing rule in the Web Service Proxy (WSP) processing policy editor and exporting it.

Follow these steps to create the assertion template:

1. Create a WSP with your WSDL, then go to the **Policy** tab and create a processing rule that enforces the behavior specified for your assertion (see Figure 2). You will export this rule later on (so note the rule name). Once you have configured the rule and tested its behavior, you are prepared to proceed to the next step.

   **Figure 2. Configuring the processing rule**

2. Go to the **Export Configuration** utility on the Control Panel as shown in Figure 3.

   **Figure 3. Export Configuration**
3. The "Export configuration and files..." option allows you to export configuration as an XML file. Choose this option and press **Next** as shown in Figure 4.

**Figure 4. Select “Export configuration and files” from the Export Wizard**

4. Select the processing rule to export (Steps 1 and 2 in Figure 5). Ensure that you are exporting all the objects required by your assertion by selecting the **Include all objects** option (Step 3). Select the **XML Config** option so the export will be produced as an XML file (Step 4). Finally, enter a name for the exported file (Step 5), and press **Next** (Step 6). The next panel provides a link to download the exported XML file.

**Figure 5. Options to use when exporting the processing rule configuration**

5. Once you have downloaded the file, you can use the **XML Firewall utility** provided in the Download section of this article to extract the rules and required objects. We have used the following command for this step:

```
curl -d @myAssertion.xcfg http://<your-appliance-hostname>:10200 | tidy -xml -i > myAssertion.xml
```
6. You may recall from Step 1 that the processing rule for this example is simple (one match action, one transform action, and one results action), so the result of Step 5 contains is two processing actions. Note that the match actions are not included in a rule export because they actually "belong" to the processing policy. The result action at the end of this rule must be removed because the policy framework automatically adds a result action after the last assertion processed by the framework for each a policy subject. In the case of this example, we are left with the action shown in Listing 5. In the case of this example, the only changes that you need to make to this action before using it in your policy mapping stylesheet is to:
   a. Provide a unique name for the actions in your assertion. Note that the same action sequence may be used many times, so action names cannot be static strings.
   b. Update the <Input> and <output> elements to use the input and output context names provided by the policy mapping framework.
   c. Add a <PolicyKey> element to enable the collection of mediation enforcement metrics. An example of each of these changes is found in the MediateMessage assertion provided in the yourco.com-example.xsl custom policy file in the Download section of this article.

Listing 5. Generating configuration for the MediateMessage assertion

```xml
<StylePolicyAction name="myService_rule_0_xform_0" ... >
  <mAdminState>enabled</mAdminState>
  <Type>xform</Type>
  <Input>INPUT</Input>
  <Transform>store:///myRequestTransform.xsl</Transform>
  <OutputType>default</OutputType>
  <Transactional>off</Transactional>
  <SOAPValidation>body</SOAPValidation>
  <SQLSourceType>static</SQLSourceType>
  <Asynchronous>off</Asynchronous>
  <ResultsMode>first-available</ResultsMode>
  <RetryCount>0</RetryCount>
  <RetryInterval>1000</RetryInterval>
  <MultipleOutputs>off</MultipleOutputs>
  <IteratorType>XPATH</IteratorType>
  <Timeout>0</Timeout>
  <MethodRewriteType>GET</MethodRewriteType>
  <MethodType>POST</MethodType>
  <MethodType2>POST</MethodType2>
</StylePolicyAction>
```

7. Once you have completed authoring the assertions for your custom policy domain, you can login to the DataPower appliance under the default domain and drop the policy mapping stylesheet into store://policies/custom. To verify if the stylesheet was parsed properly, you can check the domains supported in the UI status object as shown in Figure 6.
Now you are ready to attach the policy to the policy subjects in your WSDL and test the capability in the form of a policy assertion, instead of a manually configured processing rule.

Troubleshooting

Due to limitations inherent in XML schema validation, it may be difficult to create complex DataPower configuration without violating a policy. DataPower configuration objects can get rather complex. Additionally, it may not be immediately obvious that the configuration you generated has been rejected and is not being written to the appliance. To assist in the troubleshooting of schema violations, the domain export provided with this article includes a utility for validating the configuration generated by policy mapping stylesheets. To use the utility, take the configuration written by the dump-nodes extension function (see Listing 4) and run the following command:

```
curl --data-binary @YOUR.XML http://<your-appliance-hostname>:10300/service/mgmt/3.0
```

If the XML passes schema validation, the XML will be returned by the utility. If it fails, an error message will be returned. The schema violation will be logged in the DataPower system log as shown in Figure 7.

Figure 7. Schema validation error displayed in DataPower system log
Adding a custom widget to Business Space

WSRR supports the loading of any policy that supports the WS-Policy standard. Once loaded into WSRR, you can attach the policy to any of the objects in WSRR using the WSRR V8 Business Space user interface.

The authoring and management of custom policies can be provided in WSRR V8 Business Space by developing custom Business Space widgets. You can develop the widgets to implement a policy editor to create, retrieve, and update the policies within WSRR.

These custom widgets can be partially integrated with the existing WSRR widgets by wiring up the custom widgets to catch events generated by the WSRR widgets.

An example of such a widget is provided in the YourCoExampleWidget.zip file, which is provided in the Download section of this article. The functions included in the widget are:

- Create new or edit existing Policy Documents in the http://yourco.com/policy/standard/2012-08 domain.
- Support for all assertions defined by the example domain.
- Wiring for inbound item selected events.
- Wiring for outbound item created and updated events.

You may wish to undertake an exercise to address the following limitations:

- There is no ability to delete or transition Policy Documents.
- You cannot edit a Policy by clicking on the Policy, only the Policy Document.
- The Policy Document name and Policy name are automatically generated with UUIDs. No part of the user input name is used for the unique identifiers.
- There is no HTML formatting or CSS applied to the widget.
- There is no error handling.

See the "TODO" comments in PolicyEditor.js for all the things that need customization.

To install this widget for use in Business Space, perform the following actions:

1. In an Eclipse or Rational® Software Architect or Rational Application Developer, use the Import > existing projects into workspace option and select the YourCoExampleWidgetProjects.zip file.
2. Right-click the YourCoPolicyEditorEAR project and select Export > EAR file and enter YourCoExamplePolicyEditor.ear as the file name.
3. Create a zip file called YourCoExampleWidget.zip with the following structure:
   - catalog/catalog_YourCoPolicyEditorWidget.xml
   - ear/YourCoExamplePolicyEditor.ear
   - endpoints/
   - help/
   - spaces/
   - templates/

Notes:
• The "catalog_YourCoPolicyEditorWidget.xml" file is found in the
   YourCoPolicyEditorWidget project inside WebContent/WEB-INF.
• If you want to skip creating this ZIP, then one is provided for you with this article named
   YourCoExampleWidget.zip.

4. On your WSRR V8 server, execute the wsadmin command line tool and enter the following
   commands:
   a. $AdminTask installBusinessSpaceWidgets { -serverName
      <server name> -nodeName <node name> -widgets
      <full path to YourCoExampleWidget.zip> }
   b. $AdminConfig save
   c. exit

5. Login to the WSRR Business Space UI. You now find a new widget called
   "YourCoPolicyEditorWidget" that can be added to a space as shown in Figure 8.
   **Figure 8. Adding the example widget to a Business Space page**

6. Once added, the widget looks like Figure 9.
   **Figure 9. The example widget initial view**

7. If you click New Policy, then a pop-up dialog (Figure 10) appears that allows you to set the
   name of the policy and the values of the example policy domain elements.
   **Figure 10. Creating a new policy with the example widget**

8. If you then click Create, you see a read only view of the policy as shown in Figure 11.
Figure 11. Viewing a newly created policy with the example widget

You can wire the widget to any of the WSRR-supplied Business Space widgets by editing the wiring for the widget. For instance, if you wire it to the Service Registry Collection Widget as shown in Figure 12, then the example widget shows the details of the Policy Document selected in the collection widget and allows you to edit it by clicking Edit Policy. This can be achieved by selecting Edit Page, clicking on the drop down menu in the top right corner of the example widget, then selecting Edit Wiring.

Figure 12. Wiring the example widget to other Business Space widgets

For detailed information about developing Business Space widgets, see Developing widgets.

Conclusion

The use of out-of-the-box policy domains, WS-SecurityPolicy and WS-MediationPolicy, allows for the quick adoption of policy-based solution to enforcing common business service non-functional requirements - such as message security and traffic management - without requiring DataPower skills. However, to take full advantage of the benefits of the DataPower’s SLA policy framework, you can expand the scope of policy domains understood by the appliance beyond the capabilities provided by the built-in ones. Using custom policy domains bring the benefits of policy-based deployments vs. traditional configuration lifecycle deployments (such as increased business agility along with improved operational stability) to customer-defined processing patterns.
This key extensibility feature makes it possible for a rich set of processing patterns to be accessible using custom policy assertions (and their associated assertion parameters). Conveniently, just about anything that can be developed into a processing rule is enforceable by the policy framework using custom policy domain assertions.

With this flexibility comes great power; therefore, it is important to understand that policy assertions provide an opportunity for DataPower to respond to common business change while minimizing the operational effects of the change to other services. In other words, a key custom policy concept is about finding that right balance of "access to change" vs. "stability and control". By varying the scope and depth of each assertion's variability, each architect can control how the application of those assertions might affect a service proxy's workflow and policy test matrix. We recommend that as a general rule (as shown in this article's examples), always make assertion syntax as high-level as possible (for example, close to the original business requirements intent) without affecting flexibility for common business changes.

Finally, we hope that the new capabilities presented in this article series provided a good foundation to seriously consider adopting the policy-based approach to a better SOA governance and provisioning of service workflows in DataPower proxies. This is just an example of how the future of SOA policy management is expected to evolve over time - thus improving the consumability and scalability of SOA appliances in your enterprise to accommodate common business changes faster.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code sample</td>
<td>custom_policy_domains_downloads.zip</td>
<td>525KB</td>
</tr>
</tbody>
</table>
Resources

• For an introduction to the Web Services Policy language, see Web Services Policy 1.5 – Primer.
• To learn more about the best practices on defining policy assertions, see Web Services Policy 1.5 - Guidelines for Policy Assertion Authors.
• The Primer and Guidelines documents should not be considered as replacements. For more detailed information about WS-Policy, see:
  • Web Services Policy 1.5 – Framework specification
  • Web Services Policy 1.5 – Attachment specification
• Participate in the WebSphere Service Registry and Repository discussion forum.
• Participate in the WebSphere DataPower discussion forum.
About the authors

Thomas Burke

**Thomas Burke** is an Advisory Software Engineer on the WebSphere DataPower Appliance development team. Thomas has been on the DataPower team for Governance capability. Prior to his role as an SOA Governance Team Lead in DataPower, Thomas served as a Team Lead on the WebSphere Business Monitor team and the Architect for an early version of the IBM Mobile Portal Accelerator.

Mario E. De Armas

**Mario De Armas** is a Senior Technical Staff Member on the WebSphere DataPower Appliance development team. He has worked on many different areas and features of DataPower appliances. He is currently the Software Architect responsible for DataPower's integration with WebSphere Service Registry and Repository, Web services, WS-Policy, and SOA Governance functionality. His past DataPower projects included Technical Lead for XB60, AS1 protocol handler, B2B gateway, appliance management functionality, database integration, and user interface customization features.

Evan Jardine-Skinner

**Evan Jardine-Skinner** is a User Interface Software Developer on the Service Registry team at the IBM Software Lab in Hursley, United Kingdom. He joined IBM as a member of the Voice Systems Services team in 1998 and later used his services experiences to great effect in the development of Voice Response, mainly working on the ISDN signaling layers. He gained experience in porting IBM applications to new platforms working in the porting center, before his passion for usability drove him to join the Service Registry UI team in 2007. Evan graduated with an Honors degree in Computer Science from Bradford University, United Kingdom.

Oswaldo Gago

**Oswaldo Gago** is an Advisory Software Engineer on the WebSphere DataPower development. He is currently the lead developer for the DataPower Web Service Proxy, WSPolicy, and WebSphere Service Registry and Repository integration. Oswaldo joined IBM in 2002 as part of the IBM Text to Speech team after he
graduated Summa Cum Laude with a Master's degree in Computer Science from the University of Miami, Florida.