**Chapter 11: Proxying Web Applications with DataPower**

Add a note hereThroughout this book, we have talked about how valuable DataPower can be when fronting XML-based applications such as Web services, as well as providing an ESB for your infrastructure. Hopefully you have seen that there is no doubting its capability to provide several benefits in these scenarios, such as threat protection, protocol bridging, routing, and so on. What about all the Web applications in your organization that are exposed to external and internal-facing clients? Can DataPower provide some benefit to these? The answer is yes. DataPower can provide a tremendous amount of value to your organization by placing it in front of these applications. What that value is and how you implement it is precisely what we discuss in this chapter. Although the main use case for DataPower in this scenario is to provide AAA functionality, we do not focus on that in this chapter, as it is similar to configuring AAA policies for other backend applications which is discussed in depth later in [Chapter 16](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=2926#2926), [“AAA,”](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=2926#2926) and [Chapter 17](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=3157#3157), [“Advanced AAA.”](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=3157#3157)

**Add a note here****Web Applications Are “Different”**

Add a note hereNow that we have discussed several different types of services, you realize how easy it could be to verify request and response messages in a DataPower service by simply importing a WSDL or referencing a schema. This is simple and effective because WSDL-based applications, or Web services, are basically atomic, stateless applications and usually involve computer-to-computer interaction. This enables a describable and predictable message format and flow. For example, only operations described in the WSDL can be called with the message format described in the schema. A response message schema is also provided in the WSDL, describing the format that the response must be in. Any subsequent request to the Web service by that client is a new transaction with no dependency or ties to the previous transaction(s). Because WSDL files can be added to and referenced from registries such as UDDI or WSRR, the consumer can dynamically construct a Web service call without any human intervention.

Add a note hereDue to the nature of Web applications, things might not be that simple when proxying them because they are extremely different from the Web services model. To start with, Web application clients are usually browser-based, with a human doing the driving rather than computer-to-computer interaction. If there are two things that are unpredictable, they are browsers and humans! There are many variations of browsers, each with their own interpretation of the HTML specification and nuances. Then there are the humans at the other end—there is no telling what they might do. Between the browser’s Back and Forward buttons and the application navigation, they can cause all kinds of havoc that is entirely unpredictable.

Add a note hereAfter we get past the unpredictability of the client, we have to deal with the Web application in the back. Web applications are typically stateful, requiring the backend application to “remember” the client and session data throughout many concurrent requests. Let’s take the common shopping cart request scenario. As a person is browsing through the catalog of an online retailer, he might add certain items to a shopping cart. As he proceeds to check out, the application must remember all the items that were added to the shopping cart during the shopping experience.

Add a note hereHumans rarely want a single piece of information in a computer-consumable format; instead, they want a rich, interactive, and often graphical experience. This requires support for multiple message formats; not just HTML, but images, videos, and yes, even structured data. To make these interactions personalized and “pretty,” content rendering engines allow JavaScript, cascading stylesheets, and cookies to affect both the information seen by the user and the manner in which it is displayed. This is not even getting into the new possibilities with Ajax and Web 2.0! An attempt to represent any of this interaction in something as simple as a WSDL file would simply be ludicrous.

Add a note hereIt should now be obvious that Web applications are different from the other types of services we have been dealing with thus far. This is not intended to scare you off from ever wanting to deal with a Web application again. It is simply to make you aware that there are many things that you need to consider when configuring a DataPower service to front your Web applications. Hopefully, by the end of this chapter, you will have gained the knowledge of how to deal with and overcome these challenges.

## Why Use DataPower?

Add a note hereYou might be thinking to yourself “My applications are running fine and given all these challenges, why would I introduce DataPower into the picture?” This is a valid question. Let’s take a look at why you might want to use a DataPower device to proxy your applications and you might change your mind.

### Add a note hereThreat Protection

Add a note hereWe have discussed the many different types of threats that can potentially penetrate your Web services and other XML-based applications. Many of these threats are relevant to Web applications as well. It is possible for Web requests to contain data that is formatted as XML, where the backend application must parse and process this data. The same types of XML threats that we have discussed throughout this book apply to these applications. Web applications are also susceptible to other threats such as SQL injection and dictionary attacks. In addition to these common threats, these applications are at risk to cross-site scripting attacks. This is where malicious code is injected into Web pages that will be viewed by other users in an attempt to gain access to sensitive data. DataPower provides protection against all these types of threats with minimal configuration required.

### Add a note herePerimeter Security

Add a note hereIt is becoming increasingly popular for organizations to require that all requests to backend applications be authenticated prior to entering the trusted zones in their network. This would mean that the authentication has to take place in the DMZ. This presents a challenge due to the fact that you would never deploy an application server or Java runtime in the DMZ. This means that the application server can no longer perform the authentication for incoming requests. Because the DataPower device is a hardened, DMZ deployable appliance, it is a good candidate to provide this functionality. Because it can accept many different types of credentials, authenticate, and authorize the user using the credentials passed, and assert the identity to the backend application server in many different formats, it can provide perimeter security for various types of applications running on various application servers. This allows the DataPower service to convert one security token to another, which provides integration between application servers consuming different types of tokens. By having the DataPower service provide this AAA functionality, the burden of this resource-intensive task is taken off the application, freeing it up to perform the business logic that it is intended to do. We discuss how to implement this security in a DataPower service later in this chapter.

Add a note hereIn addition to the threat protection and perimeter security, the DataPower appliance also provides other benefits, such as SSL termination, encryption services, and URL rewriting.

## Choosing a DataPower Service

Add a note hereAs we walk through the different DataPower service types that we have to choose from, it may seem fairly obvious which service type to use in what circumstances. For example, when proxying a Web service, you would use a Web Service Proxy. When dealing with multiple protocols, you would use a Multi-Protocol Gateway (MPGW). These scenarios seem straightforward, as the name of the service type indicates its usage. In [Chapter 6](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=755#755), [“Introduction to DataPower Services,”](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=755#755) we briefly discussed a service type called a Web Application Firewall (WAF). As with the previously mentioned service types, this name seems to indicate its usage and would tell us that we would use it to proxy Web applications. This may be the case in some instances, but not in all. The WAF service type has some powerful features for proxying Web applications; however, it is not the only service type that can be used.

Add a note hereThe Web application firewall provides drop-dead simple integration of transport-layer encryption, threat protection, and standard authentication and authorization mechanisms into an existing Web application. It also provides easy definition and enforcement of security policies regarding commonly used Web application parameters such as cookies, form variables, query strings, and HTTP headers. However, for cases that require significant message structure processing, dynamic routing to various backends, or additional nonsecurity-related features, you may want to use one of the other DataPower service types, such as the MPGW. Let’s discuss some of the WAF features more specifically, and then go beyond this service to add some functionality.

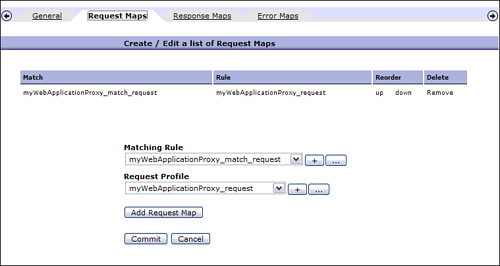
### Add a note hereWeb Application Firewall

Add a note hereAs its name implies, the WAF is used to provide a firewall, or proxy, for a backend Web application. It can be configured to perform all the AAA functionality that is found in all other service types as well as other multistep processing. Because it is specifically designed to proxy Web applications, the WAF provides a fast and convenient way to create a service by using a wizard-driven process. This wizard guides you through a step-by-step process to configure the WAF for providing AAA functionality on behalf of the backend application, which is a typical use case for this service type. When the service is created, all the service parameters default to values that would be common for most Web applications. For example, the common HTTP methods used for Web applications, such as GET, POST, and HEAD, are enabled by default. The WAF provides many added features over and above the other service types that are applicable only to Web applications. These features are

* Add a note here**Session Start and Timeout—** The WAF can set a session timeout value for a user’s browser session. When the session expires, the user can be redirected to a login page.
* Add a note here**Cookie Encryption/Signing—** The WAF can easily encrypt or sign all cookies.
* Add a note here**Name-Value Input Processing—** The WAF can configure name-value lists for request data such as headers, query parameters, form fields, and cookies. The name-value pairs can be used to filter invalid request data. The names and the values can be regular expressions that will be used to match the incoming request data. For example, if an application has a form that contains a field used for entering a numeric, five-digit ZIP code, a name-value expression can be set up for that form field with a regular expression that dictates that the value must be five characters and numeric.
* Add a note here**Cross Site Scripting (XXS) Protection—** The WAF can be configured to provide protection against cross-site scripting attacks.

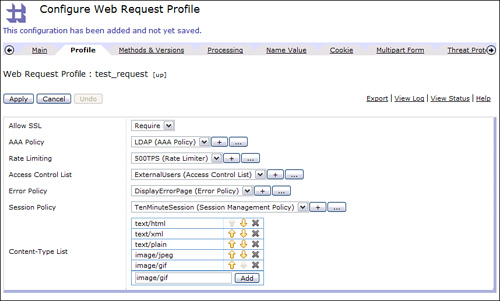
Add a note hereAs you can see, the WAF provides some nice features to enhance the threat protection and security benefits offered in other firewall types. A good use case for this service type would be for proxying a Web application providing AAA functionality and form field validation. This ensures all requests are authenticated before reaching the trusted zone in the network. Also, by enforcing this positive security model, which enables only acceptable, properly formatted form field data, the request data is sanitized, reducing the amount of exceptions thrown by the backend application due to bad data.

Add a note hereBefore you jump right into configuring a WAF, we should first discuss some differences in the WebGUI from what you have seen thus far with other service types. With WAF, the focus is no longer on message transformation and processing, so the configuration screens are significantly different. They are less graphically driven and tailored more to the quick configuration of multiple text-based security policies. In addition, much of the terminology is specific to Web applications, rather than to generic message processing. To give you some idea of these differences, take a look at Figure 11-1. This is where you configure your request, response, and error rules for your service. This is the equivalent to the Policy Editor in the other service types. The first thing you might notice is that the title of the screen refers to an Application Security Policy rather than a Processing Policy as in other services. It is called a Security Policy, which indicates that the intent of this service type is not necessarily to process the request/response message but to provide security.

[](javascript:PopImage('IMG_273','http://images.books24x7.com/bookimages/id_30903/11fig01_alt.jpg','759','404'))  
Add a note hereFigure 11-1: WAF Security Policy Editor.

Add a note hereAs we know, a Processing Policy (or Application Security Policy) is made up of rules and actions. This is also true within the WAF configuration; however, they are not called rules. Notice in Figure 11-1 that there are several tabs that reference different types of “maps,” such as Request Maps, Response Maps, and Error Maps. These are the rules that you are used to configuring in other service types.

Add a note hereEach Request/Response Map in a security policy contains a matching rule and an accompanying profile. The matching rule defines which headers, URLs, or input data will determine the profile that is executed. A profile is simply a collection of policies that will be enforced on the incoming message. These include AAA policies, session policies, protocol restrictions, and various name-value profiles. Figure 11-2 shows an example of a request profile that provides a AAA policy and many other capabilities the WAF has to offer.

[](javascript:PopImage('IMG_274','http://images.books24x7.com/bookimages/id_30903/11fig02_alt.jpg','858','517'))  
Add a note hereFigure 11-2: Web Request Profile.

Add a note hereThere may be a slight learning curve when configuring a WAF for the first time. This should not discourage you from using it as there are several added benefits when using this service type. It is ideal for providing perimeter security, form field validation, and Web-specific threat protection for Web applications.

### Add a note hereAnother Option

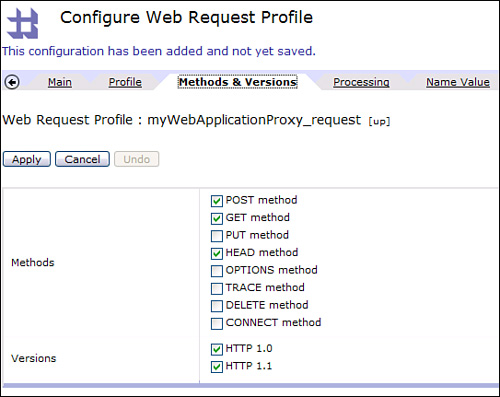
Add a note hereAs we saw in the previous section, the WAF service focuses primarily on securing your Web applications by providing threat protection, AAA functionality, form field validation, and so on. There may be times where you need to focus on processing the data in the Web application request and response, such as within a Processing Policy. In this case, you might decide to use an MPGW service type as the Web application proxy. Because the MPGW supports HTTP and HTTPS(s), it can proxy Web applications. The MPGW can provide AAA functionality, dynamic routing, multiple FSHs, header rewriting, and more. If you require that a processing rule executes on each and every transaction, this can be configured as well. Keep in mind that because the MPGW is not specifically designed for Web applications, some default configuration parameters need to be changed. These specific parameters and options are discussed in the [next section](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=1892#1892).

## Service Configuration Parameters

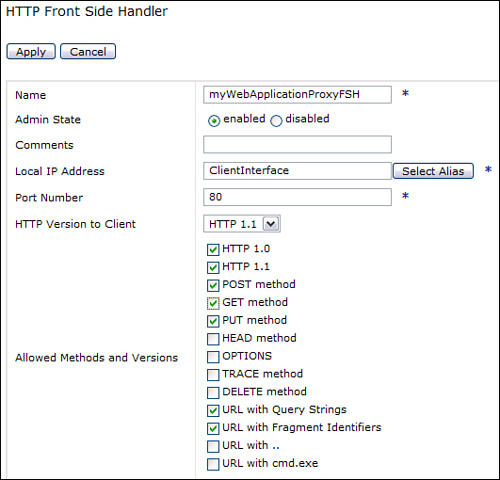
Add a note hereBecause every Web application is different, it is likely that you will have to change some of the configuration parameters in your service for each application you are proxying. In this section, we discuss some of the most common parameters that you should be aware of in the WAF and the MPGW services.

### Add a note hereMethods and Versions

Add a note hereIn a DataPower service, you can define which HTTP methods as well as HTTP versions are allowed. It is likely that you will allow only the POST, GET, and HEAD method to your Web application because they are the most commonly used. The WAF is specifically designed to proxy Web applications, so these methods are allowed by default. This can be set at the request profile level for each one configured within the service. If, for some reason, you want to disable these or enable others, this configuration screen can be found in the Methods and Versions tab in the Request Profile, as shown in Figure 11-3.

[](javascript:PopImage('IMG_275','http://images.books24x7.com/bookimages/id_30903/11fig03_alt.jpg','543','431'))  
Add a note hereFigure 11-3: WAF methods and versions.

Add a note hereIt is not likely that you will have to change these parameters for a WAF service; however, you will need to make a change for an MPGW when proxying Web applications. When an HTTP FSH is configured, it does not allow the GET method by default. This can be changed within the Front Side Handler Object(s) as shown in Figure 11-4.

[](javascript:PopImage('IMG_276','http://images.books24x7.com/bookimages/id_30903/11fig04_alt.jpg','531','510'))  
Add a note hereFigure 11-4: MPGW methods and versions.

### Add a note hereRequest and Response Type

Add a note hereBecause the MPGW can accept requests and responses in many different formats, the request and response type must be changed from the default (SOAP) when proxying Web applications to Non-XML. Because the WAF is specifically designed to proxy Web applications, this is not a concern; however, you can specify the allowable content type.

### Add a note hereFollow Redirects

Add a note hereAs a Web application responds to a client request, there are many times that the response will contain a redirect directive in the HTTP header, which indicates that the client should redirect the request to the URL specified. The client browser would act on this directive and automatically send a new request to this location. It is possible to have your DataPower service recognize these redirect responses, make a request to the location specified, and then return the response to the browser. The setting to enable and disable this feature can be found on the HTTP Options tab of the service configuration screen, which can be accessed only through the Navigation menu. For a MPGW service, this can be found via Objects→Multi-Protocol Gateway→Service name→HTTP Options. Similarly, it can be found for a WAF via Objects→[Web Application Firewall](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=1875#1875)→Service name→HTTP Options. The default for this parameter is On, which tells the service to follow the [redirects](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=1956#1956) for the client.

Add a note hereThere are a couple of reasons why you may want to turn this off. The first reason is that the redirect location may not always be accessible from the DataPower device, therefore, it should be requested by the client browser. Also, when the DataPower device performs the redirect, the URL address in the client browser never changes to the new location. This unusual behavior could cause problems if the end user wants to bookmark this address. As a general practice, it is recommended that this be turned off, allowing the browser to follow the redirects as intended.

### Add a note hereRewrite Hostnames When Gatewaying

Add a note hereWhen a browser sends a request to a Web application, it can automatically add an HTTP header named Host with a value of the host address that it is sending the request to. If the request was sent directly to the application server, this value would be the address of the application server itself. Because the application is being proxied by a DataPower service, the Host header would most likely be the address of the DataPower device, load balancer, or virtual IP that the request was sent to. As the DataPower service receives this request, it has the capability to rewrite this header to the value of the backend server that it is forwarding the request to.

Add a note hereIt is common for Web applications to use this header to build a redirect location URL to send back to the client. This is where the problem lies with having DataPower rewrite this Host header. If the DataPower service rewrites this header to the actual value of the backend server, the redirect location would be built pointing to this server and bypassing the DataPower service that proxies the application.

Add a note hereTo demonstrate this behavior, look at Figure 11-5, where a client is sending a request to the DataPower device, which is proxying a backend Web application. The service in this scenario is configured to *not* rewrite the Host header. Notice that the Host header sent in the request is the address that the request was sent to (DataPower). As the request is forwarded to the application, this header is not changed. Now when the application uses it to build a redirect location, it will use the Host header, which is the DataPower device. When the browser follows this redirect, it will again go back to the DataPower service that proxies the application.

[](javascript:PopImage('IMG_277','http://images.books24x7.com/bookimages/id_30903/11fig05_alt.jpg','709','150'))  
Add a note hereFigure 11-5: Redirect after DataPower does not rewrite the Host header.

Add a note hereFigure 11-6 shows the same example but demonstrates what would happen if the DataPower service rewrote the Host header to be the address of the backend server. Now when the redirect location is set, it is set to the address of the application server itself. This causes the browser to attempt to bypass the DataPower service and access the application directly. This is not the desired behavior.

[](javascript:PopImage('IMG_278','http://images.books24x7.com/bookimages/id_30903/11fig06_alt.jpg','707','156'))  
Add a note hereFigure 11-6: Redirect after DataPower rewrites the Host header.

Add a note hereThis is why it is recommended you set this parameter to Off in the service configuration. This setting can be found in the HTTP Options tab of the service configuration screen. As mentioned before, this screen must be accessed through the left navigation menu: Object→Multi-Protocol Gateway→Service name →HTTP Options for a MPGW service and Objects→[Web Application Firewall](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=1875#1875)→Service name →HTTP Options for a WAF. The default for this parameter is On and should be changed.

## Request Processing

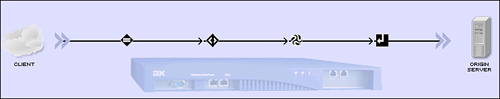
Add a note hereAs a request is sent to a Web application, it may be desirable to take some action based on data within the request. This might include query parameters, form data, or request headers. This section discusses different techniques that can be used for accessing this data.

### Add a note hereQuery Parameters and Form Data

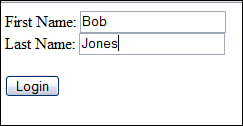
Add a note hereMuch like XML-based services, there are many cases in which you would be interested in inspecting the payload of the request to a Web application. Instead of an XML document in the payload, Web applications can contain POST data that is submitted from an HTML form. In addition to sending POST data, a request can send name value pairs in the URI, called query parameters. This data can be accessed in a DataPower service to perform tasks such a dynamic routing, logging, or even AAA.

Add a note hereAs you might have realized by now, DataPower is optimized for dealing with XML. If you can get this data into an XML nodeset, you can parse it, query it with XPath, transform it, and so on. Fortunately, there is a specific action provided to do just that. This action is called Convert Query Params to XML and is found in the list of actions within the Advanced action. You can simply add it to the Processing Policy, and it will convert the form data or query parameters to a well-formed XML nodeset.

Add a note hereLet’s look at an example to demonstrate how this works. Figure 11-7 shows a request rule that consists of a Match Rule, a Convert Query Params to XML action, and then a Transform action. As this rule executes against a request that contains POST data, we would expect the output of the Convert Query Params to XML to be a well-formed XML nodeset containing the POST data fields and values. This would then be the input to the Transform action where it can be evaluated.

[](javascript:PopImage('IMG_279','http://images.books24x7.com/bookimages/id_30903/11fig07_alt.jpg','739','147'))  
Add a note hereFigure 11-7: Convert Query Params to XML rule.

Add a note hereListing 11-1 shows a simple HTML document that, when opened with a browser, would present a form as shown in Figure 11-8, where we have entered a first name and a last name, respectively. Notice in the HTML code that there are two fields within this form. One field has a name of firstname and the other lastname.

  
Add a note hereFigure 11-8: Test Form within a browser.

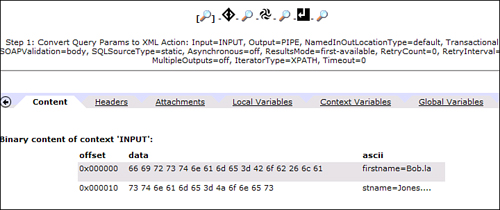
Add a note hereListing 11-1: Test Form HTML

Add a note hereFirst name: <input type="text" name="firstname"><br>

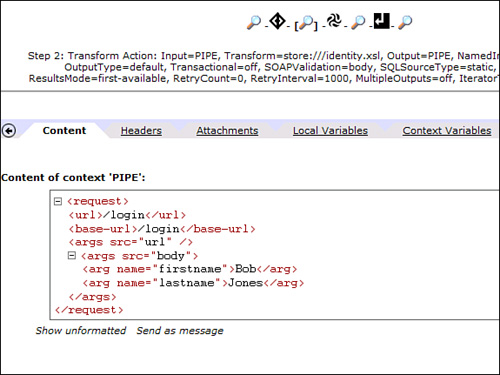
Last name: <input type="text" name="lastname">

<input type="submit" value="Submit"> </form>

Add a note hereIf we were to submit this form to the DataPower service where it would be processed by the rule shown in Figure 11-7, we would expect these two fields to be included in an XML nodeset that would be created and used as the input to our Transform action. To illustrate this, the probe in Figure 11-9 shows the request as it entered the rule in ASCII format.

[](javascript:PopImage('IMG_281','http://images.books24x7.com/bookimages/id_30903/11fig09_alt.jpg','720','303'))  
Add a note hereFigure 11-9: Probe before Convert Query Params to XML action.

Add a note hereAfter the request is processed by the Convert Query Params to XML action, the form data is converted into an XML nodeset. Figure 11-10 shows the probe with the output of this action displayed. You will notice that the output contains an XML nodeset with several elements. You can see within the <args src=“body”> element that there are two elements representing the form fields and values of the form that was submitted. This XML can now be parsed and processed by the transformation step.

[](javascript:PopImage('IMG_282','http://images.books24x7.com/bookimages/id_30903/11fig10_alt.jpg','575','431'))  
Add a note hereFigure 11-10: Probe after the Convert Query Params action.

### Add a note hereRequest Headers

Add a note hereAnother part of the request that may be of interest is the set of HTTP headers. For example, you may want to view the cookie header to view all the cookies sent within a request. It is also possible to add, remove, and append headers to the HTTP request. These functions are typically performed within an XSLT stylesheet using the DataPower extension functions and elements. The available functions and elements are

* Add a note here**dp:request-header()—** Used to read a protocol request header
* Add a note here**<dp:set-request-header>—** Used to set an HTTP request header
* Add a note here**<dp:append-request-header>—** Used to append to an existing protocol header or add a new header if the one specified does not exist
* Add a note here**<dp:remove-http-request-header>—** Used to remove the specified HTTP request header

Add a note hereFor additional information and implementation details for these extension functions, you can refer to [Part VI](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4077#4077), “[DataPower Development](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4077" \l "4077" \t "_parent),” or the DataPower extension functions document shipped with the device. Also note that there are corresponding extension functions and elements to work with the response headers.

|  |  |  |
| --- | --- | --- |
|  | Warning | Add a note hereThe remainder of this chapter discusses topics that require XSLT knowledge. If you are not familiar with XSLT programming in DataPower, please refer to [Chapter 22](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4198#4198), [“Introduction to DataPower Development,”](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4078#4078) and [Chapter 23](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4198#4198),[” Programming Within the DataPower Environment,”](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4198#4198) before proceeding.  Add a note hereAlthough XSLT knowledge is not mandatory when proxying Web applications, there may be instances when you will need this flexibility when using an MPGW for the topics that follow. |

## Response Processing

Add a note hereWithin a Web application proxy, you may need to take some action based on the response back from the application. For example, you may want to inspect the response code sent from the application server and take action on it. You may even want to redirect the user to a different location based on the response received from the application. This can all be done within a response rule. In this section, we discuss how to accomplish these tasks.

### Add a note hereResponse Codes

Add a note hereAs the response is sent back from the application, it may be required to take specific action within the DataPower service based on the response code. For example, when the backend application server is experiencing problems or is unavailable, it may send back an HTTP response code 500 with no content. Sent directly to the client, the browser displays a standard error page that is not very informative or visually stimulating. By examining the response code in the response, your DataPower service can take action when this code is returned and either serve a custom error page or redirect the client to a custom error page. This page might have the look and feel of the rest of the application and could provide a more descriptive error message that might include the helpdesk contact information and a unique error ID. This error ID could have been written to a log by the DataPower service prior to sending the redirect and could then be used by the help desk to correlate the problem call with the log entries to gain more insight into the failed transaction.

Add a note hereAs the response is returned from the application, the DataPower service sets the response code in an HTTP response header named x-dp-response-code. This value can be retrieved using the DataPower extension function dp:response-header. For example, the following line of XSLT retrieves this response code from the header and stores it in an XSLT variable named http-resp-code:

Add a note here<xsl:variable name="http-resp-code" select="dp:response-header

('x-dp-response-code')"/>

Add a note hereThis variable can then be used in further processing within the stylesheet. Setting this variable can also be accomplished via the Set Variable action.

Add a note hereIt is important to note that retrieving the response code is entirely different if you are executing an error rule. When an error is fired after the response is sent back from the request, the DataPower service stores the response code in a DataPower variable named var://service/error-protocol-response. This variable can be accessed using the DataPower extension function dp:variable(). The following line of XSLT shows how you could read the response code into an XSLT variable from within an error rule:

Add a note here<xsl:variable name="http-resp-code"

select="dp:variable('var://service/error-protocol-response')"/>

### Add a note hereRedirects

Add a note hereAs mentioned earlier, the backend application that your DataPower service is proxying may respond with a redirect directive and a location instructing the client browser to redirect the client to this new location. This redirect directive is indicated by a response code of 3XX along with a Location header indicating to the client agent the new location to send the user request. There are several redirect response codes that can be used for different reasons; however, the most common response code you will see for a redirect is a 302 accompanied by a reason phrase of Found.

Add a note hereSo what does all this mean to your DataPower service? Although it is possible to intercept a redirect response from the backend server and take action on it, such as change the Location header, it is more likely that you will be creating a new redirect and sending it back to the client from the service. For example, we discussed previously that you may want to intercept certain response codes, such as a 500, within a response rule and redirect the user to an error page. This is entirely possible by recognizing the response code, setting it to the correct redirect response code, and then setting the Location header with the new URL. Because we already know how to access the response code and how to add response headers, we can accomplish this fairly easily.

Add a note hereListing 11-2 shows an XSLT code snippet demonstrating how you might perform a redirect based on the return code from the backend server. Notice that we obtain the return code from the DataPower variable x-dp-response-code in the first line. We then check whether the first three characters are equal to 500. If so, we send a redirect back to the client by setting the applicable headers as discussed.

Add a note hereListing 11-2: XSLT Snippet to Redirect Based on Return Code

Add a note here<xsl:variable name="return\_code" select="dp:http-response-header

('x-dp-response-code')"/>

<xsl:if test="substring($return\_code,1,3) = '500'">

<dp:set-http-response-header name="'x-dp-response-code'" value="'302'"/>

<dp:set-http-response-header name="'Location'"

value="'http://www.myAppServer.com/error.html'"/>

</xsl:if>

Add a note hereListing 11-2 demonstrates how a redirect can be sent back to the client from within a response rule. Remember, this will be different if you are processing the response in an error rule because the return code is obtained and set differently, and the response headers are set differently, too. Listing 11-3 shows the XSLT that would accomplish this. Notice that in addition to setting the response code and the Location header, you must also set one additional header for the reason phrase. The reason phrase for a 302 redirect should be Found as shown. Also notice that the Location header is being set using an XSLT variable that would have been set to contain the location string.

Add a note hereListing 11-3: Setting a Redirect from an Error Rule

Add a note here<dp:set-variable name="'var://service/error-protocol-response'"

value="'302'"/>

<dp:set-variable name="'var://service/error-protocol-reason-phrase'"

value="'Found'"/>

<dp:set-variable name="'var://service/set-response-header/Location'"

value="$redir\_location"/>

Add a note hereIf, for some reason, you are proxying your Web application using an XMLFW, this will also be slightly different. When setting the response code, you cannot just set it to 302. It must include the full response code and reason phrase (for example, 302 Found).

## Cookies

Add a note hereLike it or not, cookies play a big part in Web applications and are used extensively. If you are not familiar with cookies, they are name:value pairs that are sent by the application and are stored by the browser. These cookies are then sent back to the application server in subsequent requests within one header called Cookie. There are two types of cookies that can be set. The first type of cookie is a persistent cookie. A *persistent cookie* is saved on the client computer with a timeout value. This cookie lives until it has expired and survives browser restarts and computer reboots. This type of cookie should never be used when storing sensitive data. The second type of cookie is a session cookie. A *session cookie* is set with no timeout value, which indicates to the browser that the cookie should be stored within its memory space until the browser session ends. If cookies must be used, session cookies are strongly recommended.

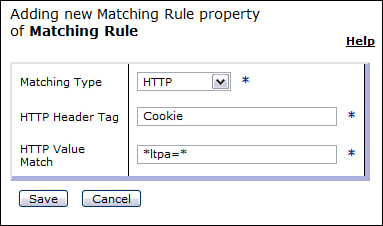
Add a note hereBecause cookies are set and sent via request and response headers, your DataPower service can read, change, and create cookies. The only difference here is that there are many cookie name:value pairs within one header. Because we already know how to handle headers within a DataPower service, let’s dive into some cookie-specific details.

### Add a note hereMatch Rules and Cookies

Add a note hereAs a request enters your DataPower service, it may be necessary to execute a particular rule in the Processing Policy upon the existence of a cookie within the request. Because all cookies sent by the browser are contained in one request header called Cookie, we can simply match on an HTTP header tag named Cookie. Now that we know we are matching on a header named Cookie, we need to supply the match criteria. We know that cookies are name:value pairs delimited by a semicolon. We may know the name part of the equation but we might not know the value, and we certainly cannot predict where in the sequence of cookies it will be.

Add a note hereFor example, let’s say we wanted to match on a request that contains a cookie named ltpa. We are not interested in the value of the ltpa cookie at this time because we are just matching on the existence of the cookie. Let’s say a request was sent containing a total of three cookies and the ltpa cookie is second in the sequence, which might look something like myName=Joe Public;**ltpa=&356ah6!9\*2**;myStatus=gold

Add a note hereNotice that the text ltpa=&356ah6!9\*2 is the cookie we are interested in. Because the only thing that we know will be in this string is ltpa=, we can use this string as the match criteria by placing a wildcard character (\*) before and after it. Figure 11-11 shows this Match Rule configuration.

  
Add a note hereFigure 11-11: Match Rule for ltpa cookie.

### Add a note hereReading Cookies

Add a note hereAfter you have matched on a cookie, you may want to parse the cookie in an XSLT stylesheet and inspect the value. To accomplish this, you can extract the header named Cookie (we know how to do this) and then extract the particular cookie you are interested in. To extract the cookie, use a regular expression. If you are not familiar with regular expressions, please refer to [Chapters 22](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4078#4078) and [23](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=4198#4198). Listing 11-4 demonstrates how to accomplish this if you are interested in the value of a cookie called ltpa. This example accomplishes this in four simple steps as follows:

1. Add a note hereRead the entire Cookie header into an XSLT variable.
2. Add a note hereCreate a regular expression that matches on the value ltpa= storing the left side of the equal sign (or the cookie name) separate from the right side of the equal sign (the cookie value) but within the same variable in separate indexes.
3. Add a note hereApply the regular expression to the entire Cookie header extracted.
4. Add a note hereAssign the value of the ltpa cookie (or right side of the equal sign) to an XSLT variable by referencing the $extracted-cookie[2] variable.

Add a note hereThe key to extracting the value of this cookie is in the regular expression that we built. In short, this regular expression means this; the \b indicates a word boundary, which is where a letter or number meets anything that is not a letter or number. Next we indicate that this word we are looking for is ‘ltpa=’ and then all characters except for the semicolon.

Add a note hereListing 11-4: Extracting a Cookie Within XSLT

Add a note here<xsl:variable name="all-cookies" select="dp:http-request-header('Cookie')" />

<xsl:variable name="cookie-regexp" select="concat('\b', 'ltpa',

'=([^;]\*)')" />

<xsl:variable name="extracted-cookie" select="regexp:match($all-cookies,

$cookie-regexp)" />

<xsl:variable name="LTPA\_Cookie" select="$extracted-cookie[2]"/>

### Add a note hereSetting Cookies

Add a note hereThere may be times when you want to set a cookie in the client browser from within your DataPower service. This cookie can then be referenced in subsequent requests as it will be sent by the browser. Cookies are sent back to the browser in a response header called Set-Cookie. The value of this header will contain the name:value pair for the cookie being set. Because we know how to set response headers within XSLT, we know how to set cookies. There is one caveat here: Because the header that is used to set a cookie is static, what if the backend application is also trying to set a cookie by sending the response header Set-Cookie? If we try to use the same extension function shown previously to set a response header, DataPower overwrites any existing Set-Cookie header. Lucky for us there is an extension function called append-response-header that appends your response header to an existing one of the same name if it exists. If there is no existing response header, a new one is created. Once the appended response header is sent back to the browser, it knows how to parse and set the cookies. For example, we could set a cookie named userName with a value of Bob to be stored by the client with one line of XSLT as follows:

Add a note here<dp:append-response-header name="'Set-Cookie'" value="'userName=Bob;'"/>

Add a note hereNotice that when we set the cookie in this example there is no mention of an expiration date. This would cause the cookie to be set as a session cookie and would expire when the client browser session ends. There are many parameters that can be set when setting a cookie, such as indicating that this cookie should only be sent over an SSL connection. This and other parameters can be found in many publications and online reference sites.

### Add a note hereRemoving Cookies

Add a note hereWe have talked about reading and writing these cookies in a DataPower service. What if you wanted to remove one from the client’s browser that you have previously set? Maybe it’s a cookie that contained a timestamp in the value and your service has decided that it has expired. You can remove it by sending another Set-Cookie response header for the same cookie but this time adding an expiration date in the past. This causes the browser to recognize that the cookie has expired and it removes it from the browser session. Listing 11-5 shows how we could remove the userName cookie from the browser that was set in the previous example.

Add a note hereListing 11-5: Removing a Cookie from the Browser

Add a note here<dp:append-response-header name="'Set-Cookie'" value="'

**Form-Based Authentication**

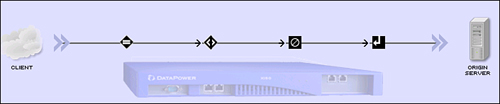
Add a note hereIn the introduction of this chapter, we mentioned that a primary use case for proxying Web applications is to provide perimeter security by means of DataPower’s AAA capabilities. We also mentioned that we will not be covering that in this chapter as it is no different from most other AAA scenarios. There is, however, one exception, and it is a common one. Web clients can be authenticated in several different ways. For example, they can use basic authentication or certificates to provide credentials to the authenticating service. As you will learn in [Chapter 16](http://www.books24x7.com/assetviewer.aspx?bkid=30903&destid=2926#2926), these two credentials can be parsed and used for authenticating the client natively within a AAA policy. One other very common way of gathering and sending credentials is called form-based authentication.

Add a note hereForm-based authentication provides a means for the application to present a customized login form to the client. This form can be decorated to match the rest of the application screens presented, including company logos and other images. These forms can be customized in any way the developer sees fit; however, the field names that are used within the form for the user name and password must match what the application server expects. When using this form of authentication within a WebSphere Application Server (WAS), these fields must be named j\_username for the username field and j\_password for the password field. This form will then POST to j\_security\_check, and the WAS security framework will then know how to extract the username and password from within the POST data.

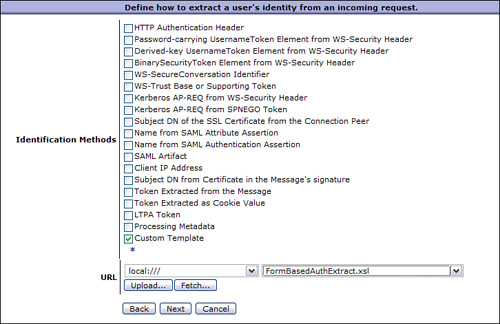
Add a note hereBecause this is a common means of authentication, we should be able to implement this within a DataPower AAA policy—and we can. It might not be an out-of-the box option seen in the AAA policy editor but it is not difficult to implement. From a AAA perspective, the only difference in form-based authentication from basic authorization is how the policy extracts the identity. After that, the authentication and authorization steps are dealing with a username and a password in each case. So once we extract the identity from the form submitted, the rest is business as usual!

Add a note hereWithin each step of the AAA policy, there are many options presented to complete the given step, such as extract identity, authenticate, and authorize. If none of the options available suits your needs, there is always a custom option that allows you to take the responsibility of that step within an XSLT stylesheet and pass the results to the next step as expected. Because there is no option in the extract identity phase for form-based authentication, we will have to customize this by extracting the identity via XSLT and pass the username and password to the authentication step as it expects it.

Add a note hereBefore we get into the AAA policy, we have to get the form data into a format that can be parsed by an XLST stylesheet. We would use the Convert Query Params to XML action as discussed earlier in this chapter. The output of this action would serve as the input to our AAA action, as shown in Figure 11-12.

[](javascript:PopImage('IMG_284','http://images.books24x7.com/bookimages/id_30903/11fig12_alt.jpg','705','147'))  
Add a note hereFigure 11-12: Form-based authentication policy.

Add a note hereWhen configuring the AAA policy, you must select the custom option in the extract identity and provide a stylesheet that does the identity extraction as shown in Figure 11-13.

[](javascript:PopImage('IMG_285','http://images.books24x7.com/bookimages/id_30903/11fig13_alt.jpg','726','471'))  
Add a note hereFigure 11-13: Custom extract identity.

Add a note hereWhen choosing the custom option in the extract identity, you take responsibility for extracting the credentials from the request and pass them to the next step in the AAA policy, which is the Authenticate step. The format in which the Authenticate step expects this data is shown in Listing 11-6.

Add a note hereListing 11-6: Format Passed to the Authentication Step

Add a note here<identity>

<entry type="custom">

<username>userid<username>

<password>password</password>

</entry>

</identity>

Add a note hereTo show how this identity extraction can be accomplished, let’s take a simple HTML form that accepts the username in a field called j\_username and the password in a field named j\_password. The HTML for this form is shown in Listing 11-7.

Add a note hereListing 11-7: Login Form HTML

Add a note here<form method="POST" action="http://myDataPowerDevice.com:6000/login">

Username: <input type="text" name="j\_username"><br />

Password: <input type="password" name="j\_password"><br />

<br />

<input type="submit" value="Login">

<input type="reset" value="Reset">

</form>

Add a note hereThe custom XSLT needs to extract the username and password from this form after it has been converted to XML by the Convert Query Params to XML action. The XML generated from this form when submitted for a user testuser and a password password is shown in Listing 11-8.

Add a note hereListing 11-8: Login Form Converted to XML

Add a note here<request>

<url>/login</url>

<base-url>/login</base-url>

<args src="url" />

<args src="body">

<arg name="j\_username">testuser</arg>

<arg name="j\_password">password</arg>

</args>

</request>

Add a note hereThe XSLT referenced in our AAA policy must now extract the username and password from this XML nodeset. Because we know the username is in element j\_username, and the password is in element j\_password, this shouldn’t be too difficult. After we extract these fields, we must then build the XML nodeset that is shown in Listing 11-7. The XSLT code to accomplish this is shown in Listing 11-9.

Add a note hereListing 11-9: Extract Identity XSLT

Add a note here<xsl:variable name="uid">

<xsl:value-of

select="/request/args[@src='body']/arg[@name='j\_username']"/>

</xsl:variable>

<xsl:variable name="password">

<xsl:value-of

select="/request/args[@src='body']/arg[@name='j\_password']"/>

</xsl:variable>

<identity>

<entry type="custom">

<username><xsl:value-of select="$uid"/></username>

<password sanitize="true"><xsl:value-of

select="$password"/></password>

</entry>

</identity>

Add a note hereNow that we have created our policy to convert the form to XML and the custom stylesheet to extract the credentials, the rest of the AAA policy can be configured as usual. The extracted credentials are passed to the Authentication step where they can be authenticated using one of the options available.

Add a note hereThere you have it! This is how you would authenticate using the very common form-based authentication method. This may seem like a lot of steps and coding at first, but it is a very common means of authenticating; this AAA policy and stylesheet are reusable for many other Web application proxies.

## Can DataPower Replace the WebSphere Application Server Plugin?

Add a note hereNow that you see all the things that DataPower can do for your Web applications, your wheels are probably turning and you are thinking “maybe this can replace other components within my topology such as the WAS plugin.” At first this may seem like a suitable replacement. You should give careful thought to this before doing such a thing. Although DataPower is powerful and efficient, it is not built to do some of the things that the WAS plugin can do. For example, the WAS plugin has some powerful caching and work load management (WLM) mechanisms that are difficult to duplicate on the DataPower device. Although the DataPower device can perform load balancing to the multiple backend servers, it is no match for the sophisticated load balancing algorithms and health check capabilities provided by the plugin. The plugin is also “application-smart,” which means that it is aware of the backend application URIs with knowledge of the application deployment descriptor. This allows the plugin to gain insight into the application for things such as which resources are protected within the application.

Add a note hereSo instead of replacing the WAS plugin with DataPower, you may want to consider moving the Web server with the plugin out of the DMZ and have DataPower placed there where it can complement the Web server and plugin.

## Summary

Add a note hereHopefully this chapter has opened your eyes to the enormous potential the DataPower device has to proxy your Web applications. We have discussed the different types of services that can be used to do this as well as the benefits of each. Many valuable patterns and techniques were discussed that can be added to your tool belt to be used when you are configuring these services, including working with Web application requests, responses, and cookies. We also demonstrated a simple, yet effective way to configure a service to perform form-based authentication for your backend application. Combining these techniques can help you in building flexible, efficient, and powerful proxies for all your Web applications.