NativeSPRequestMapHowTo

The SP software includes an implementation of the RequestMapper plugin interface that combines "native" configuration support specific to certain web servers with a portable mechanism that relies on an XML syntax for applying settings to content and works across all the supported platforms.

A more complete syntax reference to using this mechanism can be found in the NativeSPRequestMap topic. This topic will outline how to use it, show some examples, and note some potential mistakes.

Apache Use

If you're using Apache, you should consider using the native `ShibRequestSetting` command, as it's generally safer and more natural to use. If you want to use the XML syntax instead, be sure to turn on the `UseCanonicalName` Apache option to avoid leaving security holes.

General Structure

The XML-based syntax operates against the *logical* URL requested by the client, and not the *physical* path or file accessed. This is analagous to the difference between the Apache `<Location>` and `<Directory>/<Files>` distinction.

The mapper executes by breaking apart the requested URL into three parts: the **host**, **path**, and **query**. The **host** portion consists of *scheme://authority:port* (everything before the slash that starts the path). The **path** is everything after the **host** portion up to, but not including, the first '?' character, if any. If a '?' separator exists, the **query** portion consists of all the decoded parameters found after that point.

Each portion is then matched against the elements inside the `<RequestMap>` in order to locate the "deepest" matching element, which is then used to derive the content settings to apply to the request.

The structure of the map, as with XML in general, is of a tree, with elements nested inside of other elements in a parent child relationship. The mapping process walks this tree by evaluating each set of siblings and, when a match is found, descending into the matching element to evaluate its child elements, and so on. The information from the requested URL that matches is "consumed" each time the algorithm descends into a matching element, and only the remaning portion, if any, is available for further matching. Eventually, nothing is left to match against, no child elements remain, or none of them match. At that point, the process stops with the last matched element and it forms the bottom of the tree to use.

To see how this works in practice, consider this moderately complex example. No actual content settings are shown, as would be found in a real example, to emphasize the matching process.

```
<RequestMap>   <!-- A -->
    <Host name="sp.example.org">   <!-- B -->
        <Path name="/"/>        <!-- Do NOT do this, it will be ignored! -->
        <Path name="secure"/>      <!-- C -->
        <Path name="admin">        <!-- D -->
            <Path name="secure"/>  <!-- E -->
        </Path>
        <Path name="admin/badexample"/>  <!-- Do NOT do this, it will be ignored! -->
        <Path name="combined/path"/>  <!-- F -->
    </Host>
    <Host scheme="https" name="internal.example.org"/>   <!-- G -->
</RequestMap>
```

Assuming that the web server is appropriately configured, the table below shows which element (labeled in the XML comments above as A-G) each input URL will map to.

Request URL	Maps to...	Notes
https://internal.example.org/anything	G	
http://internal.example.org/anything	A	the scheme is http, not https.
http://sp.example.org/stuff	B	the path portion doesn't match
https://sp.example.org/secure/anything	C	
https://sp.example.org/admin/stuff	D	

https://sp.example.org/admin/secure/anything	E	
https://sp.example.org/combined/stuff	B	the path portion doesn't match
https://sp.example.org/combined/path/anything	F	

General Tips

Note in the example above that none of the `<Path>` elements contain leading or trailing slash characters. Such characters will be stripped from the configuration and ignored, so they are insignificant.

Also note the element with a single slash labeled with a warning. An empty pathname will be ignored, and does **NOT** mean "anything on the parent host". Rather, the `<Host>` element is applied by default to any paths that don't match a child element. You don't need to specify an empty child path, in other words, to match an entire host.

However, it is allowable to include a slash **inside** a path expression in order to "shortcut" an expression intended to match only a subpath. This is shown in the example. It is equivalent to expand such an expression out into a set of nested `<Path>` elements representing the URL tree. Embedding the slash is just a shorthand.

Overlapping Siblings

Note the comment above warning about the "admin/badexample" element. This is the most common mistake made and is an example of "overlapping siblings". The bad element contains a pathname that overlaps in the URL tree with an earlier-declared sibling element (D). They are siblings in XML terms, elements with the same immediate parent element (B).

The implementation does not allow siblings to "overlap" in the URL tree, and it will ignore the overlapping element, producing unexpected results.

Matching Elements

Each portion of the request URL (host, path, query) is matched against a distinct set of elements. The XML schema is such that the top level matching elements **MUST** be host-based, using the `<Host>` or `<HostRegex>` elements. The matching element can then contain rules for either path or query-based matching as described below.

Host Matching

The matching process begins by evaluating the **host** portion of the request against the supplied `<Host>` elements, followed by any `<HostRegex>` elements.

Note that the `<Host>` can match a particular hostname for both http and https requests to the default ports (80 and 443 respectively) using a syntax like:

```
<Host name="sp.example.org"/>
```

Alternatively, you can restrict matches to a specific scheme and/or port by adding those options:

```
<Host scheme="https" name="sp.example.org"/>
```

Canonicalization and Virtualization

The host matching process is highly dependent on the ability to canonically derive a scheme, hostname, and port for the request. This is complicated by two factors: non-canonical names and virtualization.

When names aren't canonical, the server name supplied by the client dictates the name seen by the SP software. Since a client is free to locally map any name to a host address, this means the mapping process would be useless, and therefore a canonical value has to be set for each virtual host.

Virtualization occurs when the physical web server settings for port or scheme don't match the logical values seen by the client. Mapping should be based on the client's view of the URL space, and therefore the web server needs to pass the logical values to the SP instead of the physical values.

Most web servers either do not support canonical naming (IIS), or allow it to be disabled (e.g., Apache). When using the XML-based map with Apache, the `UseCanonicalName` option needs to be enabled, as mentioned earlier. IIS supports neither canonical naming nor virtualization. Instead, the SP's own configuration includes a section to supply canonical scheme, hostname, and port information corresponding to each IIS web site using the `<ISAPI>` element.

Path Matching

If a matching host-based element is identified, then its children are used to perform path-based matching by looking for `<Path>` elements, followed by any `<PathRegex>` elements.

As discussed above, a single `<Path>` element can contain one or more path "segments" separated by a forward slash. Leading or trailing slashes are ignored. A multi-segment element is equivalent to nesting each segment inside its parent. In other words, this:

```
<Path name="one/two/three"/>
```

is equivalent to this:

```
<Path name="one">
    <Path name="two">
        <Path name="three"/>
    </Path>
</Path>
```

The requested URL's **path** portion is divided into segments and processing proceeds by looking for matching segments. As matches are found, the matched segment is discarded and the process descends into the matching element and then the same process is repeated, until no further matches are found.

The matching process is case-insensitive, though this is a somewhat unreliable assumption if Unicode paths are involved.

Query Matching

Once the most-specific path match is identified (if any), the final step is to look for child `<Query>` elements to match against the decoded query string parameters. Absent such elements, the query string information is **NOT** used during path matching and ignored.

Note that query matching is case sensitive (because the CGI specification does not imply otherwise), which may lead to problems with many applications if you fail to match something but the underlying application interprets the same parameter(s) with arbitrary case.