DynamicHTTPMetadataProvider

The `DynamicHTTPMetadataProvider` fetches entity metadata just-in-time from a remote HTTP server. The metadata request URL is constructed by applying a transform to the `entityID`. The transform strategy is configured in a child element.

Metadata is cached in memory subject to a complex set of interacting settings and the cache indicators within the metadata itself, and also can be saved to disk and reloaded back into memory at reload or startup time to restore the state of the cache. This isn't a fully redundant safety net but can be used as part of an overall strategy to reduce the risk of relying on remote sources in real-time. Ultimately, remote sources have to be bulletproof or there will be outages. This can be mitigated but not fully eliminated as a risk.

Use this provider with remote metadata

The `DynamicHTTPMetadataProvider` is used with *remote metadata*. See the MetadataManagementBestPractices topic for more information.

Contents

Schema Names and location

The `<MetadataProvider>` element and the type `DynamicHTTPMetadataProvider` are defined by the `urn:mace:shibboleth:2.0:metadata` schema, which can be located at http://shibboleth.net/schema/idp/shibboleth-metadata.xsd.

Attributes

Any of the Common Attributes, the Dynamic Attributes, or the HTTP Attributes may be configured.

Common Attributes

The following attributes are required on **all** metadata provider types:

Name	Type	Default	Description
`id`	String	required	Identifier for logging, identification for command line reload, etc.
`xsi:type`	String	required	Specifies the exact type of provider to use (from those listed above, or a custom extension type).

The following attributes are common to all metadata provider types except the `ChainingMetadataProvider` type:

Name	Type	Default	Description
`requireValidMetadata`	Boolean	true	Whether candidate metadata found by the resolver must be valid in order to be returned (where validity is implementation specific, but in SAML cases generally depends on a `validUntil` attribute.) If this flag is true, then invalid candidate metadata will not be returned.

`failFastInitialization`	Boolean	true	Whether to fail initialization of the underlying MetadataResolverService (and possibly the IdP as a whole) if the initialization of a metadata provider fails. When false, the IdP may start, and will continue to attempt to reload valid metadata if configured to do so, but operations that require valid metadata will fail until it does.
`sortKey`	Integer		Defines the order in which metadata providers are searched (see below), can only be specified on top level `<MetadataProvider>` elements.
The following are advanced settings supporting a new low-level feature allowing metadata lookup by keys other than the unique entityID and are rarely of use to a deployer.			
`criterionPredicateRegistryRef` 3.3	Bean ID		Identifies the a custom `CriterionPredicateRegistry` bean used in resolving predicates from non-predicate input criteria.
`useDefaultPredicateRegistry` 3.3	Boolean	true	Flag which determines whether the default `CriterionPredicateRegistry` will be used if a custom one is not supplied explicitly.
`satisfyAnyPredicates` 3.3	Boolean	false	Flag which determines whether predicates used in filtering are connected by a logical 'OR' (true) or by logical 'AND' (false).

Dynamic Attributes

The following attributes are common to all dynamic metadata providers (i.e., `DynamicHTTPMetadataProvider` and `LocalDynamicMetadataProvider`):

Name	Type	Default	Description
`parserPoolRef`	Bean ID	shibboleth.ParserPool	Identifies a Spring bean for the XML parser used to parse metadata. Generally should not be changed.
`taskTimerRef`	Bean ID		Identifies a Spring bean containing a Java `TaskTimer` used to schedule reloads. When not set, an internal timer is created. Generally should not be changed.
`refreshDelayFactor`	Real Number (strictly between 0.0 and 1.0)	0.75	A factor applied to the initially determined refresh time in order to determine the next refresh time (typically to ensure refresh takes place prior to the metadata's expiration). Attempts to refresh metadata will generally begin around the product of this number and the maximum refresh delay.
`minCacheDuration`	Duration	PT10M (10 minutes)	The minimum duration for which metadata will be cached before it is refreshed.
`maxCacheDuration`	Duration	PT8H (8 hours)	The maximum duration for which metadata will be cached before it is refreshed.
`maxIdleEntityData`	Duration	PT8H (8 hours)	The maximum duration for which metadata will be allowed to be idle (no requests for it) before it is removed from the cache.
`removeIdleEntityData`	Boolean	true	Flag indicating whether idle metadata should be removed.
`cleanupTaskInterval`	Duration	PT30M (30 minutes)	The interval at which the internal cleanup task should run. This task performs background maintenance tasks, such as the removal of expired and idle metadata.
`persistentCacheManagerRef` 3.3	Bean ID		The optional manager for the persistent cache store for resolved metadata. On metadata provider initialization, data present in the persistent cache will be loaded to memory, effectively restoring the state of the provider as closely as possible to that which existed before the previous shutdown. Each individual cache entry will only be loaded if 1) the entry is still valid as determined by the internal provider logic, and 2) the entry passes the (optional) predicate supplied via `initializationFromCachePredicateRef`.
`persistentCacheManagerDirectory` 3.3	File specification		The directory used for an internally-constructed filesystem-based persistent cache. This is a convenience parameter to avoid specifying a full bean via `persistentCacheManagerRef`. This option will be ignored if `persistentCacheManagerRef` is specified.
`persistentCacheKeyGeneratorRef` 3.3	Bean ID	internal default instance	Identifies a Spring bean for a `Function` which generates the string key used with the cache manager. The default implementation produces the lower-case hex-encoded SHA-1 digest of the entityID of the `EntityDescriptor`.
`initializeFromPersistentCacheInBackground` 3.3	Boolean	true	Flag indicating whether should initialize from the persistent cache in the background. Initializing from the cache in the background will improve IdP startup times.

`backgroundInitializationFromCacheDelay` 3.3	Duration	PT2S (2 seconds)	The delay after which to schedule the background initialization from the persistent cache when `initializeFromPersistentCacheInBackground=true`.
`initializationFromCachePredicateRef` 3.3	Bean ID	an "always true" predicate	Identifies a Spring bean for an optional `Predicate` which determines whether a given entity should be loaded from the persistent cache at resolver initialization time.

Configuring the Dynamic Attributes

Configure the Dynamic Attributes for the desired cache behavior. In particular, the `minCacheDuration` attribute and/or the `maxCacheDuration` attribute should be adjusted based on the life cycle of the metadata. Note that the `cacheDuration` attribute in metadata (if any) contributes to the overall cache behavior.

HTTP Attributes

The following HTTP attributes are exclusive to the `DynamicHTTPMetadataProvider` type:

Name	Type	Default	Description
`maxConnectionsTotal` 3.3	Integer	100	The maximum total number of simultaneous connections allowed by the HTTP client's connection pool manager. This attribute is incompatible with `httpClientRef`.
`maxConnectionsPerRoute` 3.3	Integer	100	The maximum number of simultaneous connections per route allowed by the HTTP client's connection pool manager. This attribute is incompatible with `httpClientRef`.
`supportedContentTypes`	List of String (comma-separated)	"application/samlmetadata+xml, application/xml, text/xml"	The MIME types supported by this provider when requesting metadata from the HTTP server. The `Content-Type` response header is validated against this list. This value cannot be specified as a bean property.

The following attributes are common to all HTTP metadata providers (i.e., `DynamicHTTPMetadataProvider` and `FileBackedHTTPMetadataProvide`).

An HTTP metadata provider includes a default implementation of the `org.apache.http.client.HttpClient` interface. The attributes in the following subsections control the behavior of the default HTTP client. To override the default client implementation, configure the following attribute:

Name	Type	Default	Description
`httpClientRef`	Bean ID		A reference to an externally defined Spring bean that specifies an `org.apache.http.client.HttpClient` object. This attribute conflicts with and overrides **all** of the HTTP attributes. See the HttpClientConfiguration topic for more information.

Use of the `httpClientRef` attribute precludes the use of any and all of the HTTP attributes in the following subsections.

HTTP Connection Attributes

The following attributes apply to the HTTP connections obtained and managed by an HTTP metadata provider:

Name	Type	Default	Description
`connectionRequestTimeout` 3.3	Duration	Depends on the provider type	The maximum amount of time to wait for a connection to be returned from the HTTP client's connection pool manager. Set to `PT0S` to disable. This attribute is incompatible with `httpClientRef`.
`connectionTimeout` 3.3	Duration	Depends on the provider type	The maximum amount of time to wait to establish a connection with the remote server. Set to `PT0S` to disable. This attribute is incompatible with `httpClientRef`.
`requestTimeout`	Duration	Depends on the provider type	**DEPRECATED**: Use `connectionTimeout` instead.
`socketTimeout` 3.3	Duration	Depends on the provider type	The maximum amount of time to wait between two consecutive packets while reading from the socket connected to the remote server. Set to `PT0S` to disable. This attribute is incompatible with `httpClientRef`.

HTTP Security Attributes

The following security-related attributes apply to any HTTP metadata provider:

Name	Type	Default	Description

disregardTLSCertificate	Boolean	false	If true, no TLS certificate checking will take place over an HTTPS connection. This attribute is incompatible with httpClientRef. (Be careful with this setting, it is typically only used during testing. See the HttpClientConfiguration topic for more information.)
disregardSslCertificate	Boolean	false	**DEPRECATED**: Use disregardTLSCertificate instead.
basicAuthUser	String		**DEPRECATED**: Use httpClientSecurityParametersRef instead.
basicAuthPassword	String		**DEPRECATED**: Use httpClientSecurityParametersRef instead.
tlsTrustEngineRef 3.1	Bean ID		**DEPRECATED**: Use httpClientSecurityParametersRef instead.
httpClientSecurityParametersRef 3.3	Bean ID		A reference to an externally defined Spring bean that specifies an org.opensaml.security.httpclient.HttpClientSecurityParameters instance, which consolidates all HTTP security parameters including advanced TLS usage. This attribute conflicts with and overrides any explicit TrustEngine implementation configured as an inline <TLSTrustEngine> element. See the HttpClientConfiguration topic for more information.

HTTP Proxy Attributes

The following attributes configure an HTTP proxy for use with an HTTP metadata provider:

Name	Type	Default	Description
proxyHost	String		The hostname of the HTTP proxy through which connections will be made. This attribute is incompatible with httpClientRef.
proxyPort	String		The port of the HTTP proxy through which connections will be made. This attribute is incompatible with httpClientRef.
proxyUser	String		The username used with the HTTP proxy through which connections will be made. This attribute is incompatible with httpClientRef.
proxyPassword	String		The password used with the HTTP proxy through which connections will be made. This attribute is incompatible with httpClientRef.

HTTP Caching Attributes

The following attributes configure an HTTP cache on an HTTP metadata provider:

Name	Type	Default	Description
httpCaching	"none", "file", or "memory"	Depends on the provider type	The type of low-level HTTP caching to perform. There are three choices: • "none" indicates the HTTP response is not cached by the client library • "file" indicates the HTTP response is written to disk (but will not survive a restart) • "memory" indicates the HTTP response is stored in memory This attribute is incompatible with httpClientRef and its value may not be specified as a bean property. Some metadata providers, most notably the reloading "batch-oriented" providers, implement HTTP caching at a higher layer and tend to work best with httpCaching="none".
httpCacheDirectory	String		If httpCaching="file", this attribute specifies where retrieved files are to be cached. This attribute is incompatible with httpClientRef.
httpMaxCacheEntries	Integer	"memory": 50 "file": 100	The maximum number of responses written to cache. This attribute is incompatible with httpClientRef.
httpMaxCacheEntrySize	Integer	"memory": 1048576 (1MB) "file": 10485760 (10MB)	The maximum response body size that may be cached, in bytes. This attribute is incompatible with httpClientRef.

Configuring the HTTP Connection Attributes

For a DynamicHTTPMetadataProvider, the HTTP Connection Attributes have aggressive default timeout values:

Name	Default
connectionRequestTimeout	PT5S (5 seconds)
connectionTimeout	PT5S (5 seconds)

`socketTimeout`	PT5S (5 seconds)

These may be tightened further if desired.

Configuring the HTTP Caching Attributes

By default, a `DynamicHTTPMetadataProvider` caches metadata in memory (`httpCaching="memory"`). The default values of the HTTP Caching Attributes are optimized for numerous, relatively small metadata files (i.e., single entity descriptors).

A file cache is volatile

A file cache will not survive a restart and so there is little (if any) benefit in overriding the default in-memory caching strategy.

Child Elements

Any of the following child elements may be specified, in the specified order (i.e. filters must appear first, then optionally a trust engine, and finally one of the request construction elements.

Name	Cardinality	Description
`<MetadataFilter>`	0 or more	A metadata filter applied to candidate metadata as it flows through the metadata pipeline
`<TLSTrustEngine>` 3.1	0 or 1	A custom `TrustEngine` used to evaluate TLS server certificates. This element conflicts with and is overridden by the `httpClientSecurityParametersRef` attribute.
`<MetadataQueryProtocol>`	0 or 1	Constructs the metadata request URL based on the requirements of the Metadata Query Protocol
`<Template>`	0 or 1	Constructs the metadata request URL by means of a simple transform based on substitution
`<Regex>`	0 or 1	Constructs the metadata request URL by means of a complex transform based on a regular expression

The `<MetadataFilter>` child element is common to all metadata providers while the <TLSTrustEngine> child element is common to all HTTP metadata providers. The remaining child elements are exclusive to the `DynamicHTTPMetadataProvider` type.

At most one of the `<MetadataQueryProtocol>`, `<Template>`, or `<Regex>` child elements is allowed. If none are configured, the provider constructs the metadata request URL directly from the `entityID`. This corresponds to the "well-known location" mechanism defined in the SAML 2.0 Metadata specification, section 4.1.

Don't forget to configure a child element

If you forget to configure a child element, the provider will default to the *well-known location* strategy. This constrains the `entityID` to be an URL (not an URN) but the provider does not check the URL scheme. If the scheme on the `entityID` is "http:", the metadata exchange will be vulnerable to a man-in-the-middle attack. For this reason, the well-known location strategy should be avoided in most cases.

<MetadataQueryProtocol> child element

If the `<MetadataQueryProtocol>` child element is used, the metadata request URL is constructed according to the SAML Profile for the Metadata Query Protocol, which itself is based on the Metadata Query Protocol specification. The content of the `<MetadataQueryProtocol>` child element will be used as the "Base URL" defined in that specification.

The `<MetadataQueryProtocol>` child element has the following optional attribute:

Name	Type	Default	Description
`transformRef`	Bean ID		A reference to a transform function for the `entityID`. If used, the child element must be empty.

The `transformRef` attribute may be used if (and only if) the child element is empty (i.e., it has no content).

<Template> child element

If the `<Template>` child element is used, the metadata request URL is constructed by means of a simple transform. Specifically, the value of the `entityII`
 is substituted into the template parameter "`${entityID}`".

The `<Template>` child element has the following attributes:

Name	Type	Default	Description

`encodingStyle` 3.4	"none", "form", "path", or "fragment"	"form"	Determines whether and how the entityID value will be URL encoded prior to replacement. Allowed values are: • "none" : No encoding is performed. • "form" : Encoded using URL form parameter encoding (for query parameters). • "path" : Encoded using URL path encoding. • "fragment" : Encoded using URL fragment encoding. The precise definition of these terms is defined in the documentation for the methods of the Guava library's UrlEscapers class.
`encoded`	Boolean	true	**Deprecated. Use 'encodingStyle instead as of v3.4.** If the element contains an `encoded` attribute set to "false", the value will be replaced directly, otherwise it will be URL form encoded.
`transformRef`	Bean ID		A reference to a transform function for the `entityID`. If used, the child element must be empty.
`velocityEngine`	Bean ID	shibboleth.VelocityEngine	This attribute may be used to specify the name of the Velocity engine defined within the application.

The `transformRef` attribute may be used if (and only if) the child element is empty (i.e., it has no content).

<Regex> child element

If the `<Regex>` child element is used, the metadata request URL is constructed by means of a complex transform. The `entityID` value is first matched against the regular expression contained in the `<Regex>` element's `match` attribute. Then, the `<Regex>` element's content is treated as a replacement expression to run based on the results of the match.

The `<Regex>` child element has the following required attribute:

Name	Type	Default	Description
`match`	String	required	A regular expression against which the `entityID` is evaluated.

Note that only numeric/positional group references (e.g., `$1`) are supported.

Examples

A typical use case is to load entity metadata dynamically from a metadata query server (i.e., a server that supports the Metadata Query Protocol). Here is a complete example:

Load entity metadata from a remote MDQ server

```
<!--
    Load entity metadata from a remote HTTP server via the Metadata
    Query Protocol: https://github.com/iay/md-query

    The sample configuration below implicitly formulates a Metadata Query
    Protocol URL from the given base URL. For example, if the entityID is
    https://sso.example.org/sp, the provider will request the following
    resource:

      https://mdq.example.org/global/entities/https%3A%2F%2Fsso.example.org%2Fsp

    The sample configuration below assumes: (1) the top-level element of
    the XML document is signed; (2) the top-level element of the XML
    document is decorated with a validUntil attribute; (3) the validity
    interval is two weeks (P14D) in duration; and (4) the server conforms
    to the Metadata Query Protocol specification.

    The metadata is cached for efficiency. The minCacheDuration attribute
    (default: PT10M) and the maxCacheDuration attribute (default: PT8H)
    strongly influence the life cycle of metadata in the local cache. (Any
    cacheDuration and validUntil attributes in the metadata itself also
    influence the behavior of the local cache.) The goal is to avoid needless
    interaction with the HTTP server. To achieve this goal, you need to
    understand the life cycle of the metadata on the server. For this reason,
    it is best to ask your federation operator for specific recommendations.

    The HTTP Connection Attributes include the connectionRequestTimeout
    attribute (default: PT5S), the connectionTimeout attribute (default: PT5S),
    and the socketTimeout attribute (default: PT5S). The default values of these
    attributes are overridden in the example below.
-->
<MetadataProvider id="DynamicEntityMetadata" xsi:type="DynamicHTTPMetadataProvider"
                  connectionRequestTimeout="PT2S"
                  connectionTimeout="PT2S"
                  socketTimeout="PT4S">
     <!--
        Verify the signature on the root element of the metadata
        using a trusted metadata signing certificate.
    -->
    <MetadataFilter xsi:type="SignatureValidation" requireSignedRoot="true"
        certificateFile="%{idp.home}/credentials/mdq-cert.pem"/>

    <!--
        Require a validUntil XML attribute on the root element and
        make sure its value is no more than 14 days into the future.
    -->
    <MetadataFilter xsi:type="RequiredValidUntil" maxValidityInterval="P14D"/>

    <!-- Specify the base URL for the Metadata Query Protocol -->
    <MetadataQueryProtocol>https://mdq.example.org/global/</MetadataQueryProtocol>

</MetadataProvider>
```

Note that the `<MetadataQueryProtocol>` child element encodes the base URL of the Metadata Query Protocol. For example, consider the following child element:

Using a <MetadataQueryProtocol> child element

```
<!-- Specify the base URL for the Metadata Query Protocol -->
<MetadataQueryProtocol>https://mdq.example.org/global/</MetadataQueryProtocol>
```

The previous `<MetadataQueryProtocol>` child element is equivalent to the following `<Template>` child element:

Using a <Template> child element

```
<!--
    The Template element specifies a simple template with a single parameter.
    By default, the entityID is percent-encoded before substitution.
-->
<Template>https://mdq.example.org/global/entities/${entityID}</Template>
```

The above configuration **explicitly** formulates an MDQ protocol URL. This example is for illustration purposes only. If the server supports the Metadata Query Protocol, a `<MetadataQueryProtocol>` child element should be used instead. This intentionally hides the details of the Metadata Query Protocol.

Finally, here is an example of the well-known location strategy:

Example of Well-Known Location

```
<MetadataProvider id="WellKnownEntityMetadata" xsi:type="DynamicHTTPMetadataProvider">

    <!--
      Use the well-known location strategy to get SP metadata. The
      entityID is not configured here; it is determined from the
      AuthnRequest's Issuer element, as sent by the requester.

      In this case, the entityID MUST be in the form of a URL (rather
      than a URN). It is STRONGLY RECOMMENDED that https URLs be used
      to protect against man-in-the-middle attacks.
    -->
</MetadataProvider>
```

Frequently Asked Questions

What does “dynamic” mean?

A `DynamicHTTPMetadataProvider` fetches entity metadata as needed. We say that *the IdP queries for SP metadata just-in-time*.

Compare this to a `FileBackedHTTPMetadataProvider` that batch loads all of the entity descriptors in a metadata file whether or not the individual entity descriptors are actually needed. In contrast, a `DynamicHTTPMetadataProvider` loads exactly those entities that are needed—no more, no less. In this sense, a `DynamicHTTPMetadataProvider` is much more efficient.

OTOH, all metadata query protocols are synchronous protocols by definition. Basically the IdP is blocked until it obtains the metadata it needs.

How does metadata query work?

When an IdP receives a SAML protocol request from a particular SP, the IdP must first obtain entity metadata for that SP. If the IdP has no such metadata in its possession, metadata resolution proceeds sequentially according to a configured chain of metadata providers. Upon encountering a `DynamicHTTPMetadataProvider` in the chain, the IdP consults an HTTP client that acts as an intermediary between the IdP and the query server.

The HTTP client implements a shared HTTP cache. (RFC 7234) If the desired metadata is already cached, and the stored response is fresh, the client immediately returns the cached metadata to the IdP. Otherwise the client issues an HTTP request to the query server. Upon receiving a response from the server, the client caches the response and finally returns the metadata to the IdP.

In either case, the IdP parses the metadata and applies any metadata filters configured on the `DynamicHTTPMetadataProvider`. The metadata that ultimately emerges from the configured metadata pipeline is cached locally (in memory) for future use.

The next time the IdP receives a SAML protocol request from this SP, it again traverses the chain of providers until it encounters the `DynamicHTTPMetadataProvider`. This time, however, the IdP does not bother to consult the HTTP client since the needed metadata is in the IdP’s local cache.

How long does the metadata remain in the IdP’s local cache?

The IdP’s local cache is governed by the Dynamic Attributes. In particular, the `minCacheDuration` and `maxCacheDuration` attributes strongly influence the life cycle of metadata in the local cache. Any `cacheDuration` and `validUntil` attributes in the metadata itself also influence the behavior of the local cache.

Does the HTTP client cache the response in memory?

Yes, by default the HTTP client caches responses in memory (`httpCaching=”memory”`). Consequently, two copies of each entity descriptor reside in memory, one managed by the HTTP client and another one managed directly by the IdP.

The HTTP client may be configured for file caching but a file cache will not survive a restart so the overall benefit of file caching is reduced. In most cases, a memory cache is preferred, at least for systems with adequate memory,

Does the HTTP client support HTTP conditional GET?

Yes, the HTTP client supports HTTP conditional GET (RFC 7232) for optimal performance but the inner workings of the HTTP client are opaque to the IdP. If the IdP does in fact consult the HTTP client, and the client returns metadata to the IdP, the IdP blindly parses the metadata and applies the metadata filters. There are no optimizations implemented on the IdP side.

What if the metadata query server is down or unavailable?

When the HTTP client sends an HTTP request to a metadata query server, the SAML protocol exchange is blocked until a response is received from the server and returned to the IdP. If the client reports a failed request, the IdP continues with the next provider in the configured chain of providers. If the offending `DynamicHTTPMetadataProvider` is the last provider in the chain, metadata resolution fails.

What can I do to minimize the impact of metadata query?

There are at least three things you can do to help minimize the impact of metadata query:

1. Configure `minCacheDuration` and/or `maxCacheDuration`
2. Configure the HTTP Connection Attributes
3. Configure a robust chain of metadata providers

As noted above, the `minCacheDuration` and `maxCacheDuration` attributes strongly influence the life cycle of metadata in the local cache. The goal is to avoid needless interaction with the HTTP server. To achieve this goal, you need to understand the life cycle of the metadata on the server. For this reason, it is best to ask your federation operator for specific recommendations.

OTOH, the federation operator may influence the life cycle of metadata in the IdP’s local cache by including a `cacheDuration` attribute in the metadata itself. In that case, the deployer has fewer configuration options to consider, by design.

The HTTP Connection Attributes include the following attributes:

1. `connectionRequestTimeout` (default: `PT5S`): The maximum amount of time to wait for a connection to be returned from the HTTP client's connection pool manager.
2. `connectionTimeout` (default: `PT5S`): The maximum amount of time to wait to establish a connection with the remote server.
3. `socketTimeout` (default: `PT5S`): The maximum amount of time to wait between two consecutive packets while reading from the socket connected to the remote server.

As noted above, each of these attributes defaults to 5 seconds. You may want to tighten these timeout values further, depending on what you know about the route to the server or the server itself.

Regardless of the IdP configuration or the service-level agreement you have with the server operator, things will go wrong. One thing you can do to hedge your bets is to deploy a local query server as backup. Alternatively, one or more high-value SPs can be pre-loaded into memory.