HttpClientConfiguration

Overview

The IdP software uses the Apache HttpClient library more or less anywhere that this functionality is required, which for most deployers is confined to obtaining metadata from remote sources. The necessary settings to control the behavior of the metadata client code can be handled directly in the *metadata-providers.xml* file in most cases, so this topic is primarily a reference for people who have very advanced needs or are using other components and features that make use of the client.

Some of the components that require or at least support the injection of a custom client bean include:

- loading service configuration resources from an HTTP server (HTTPResource)
- advanced/custom configuration of remote metadata sources (FileBackedHTTPMetadataProvider, DynamicHTTPMetadataProvider)
- reporting of metrics via an HTTP collector (MetricsConfiguration)
- a forthcoming HTTPDataConnector for web service access in the attribute resolver

Even then, many basic needs can be met with a set of built-in Spring beans and properties (described below) that provide basic client functionality, at the cost of "global" behavior (meaning all components would be relying on common settings).

You might need to dive into this topic further if you want to finely tune settings for different situations or if you have advanced security requirements that go beyond default behavior. A common example would be if you want to control TLS server authentication at a finely-grained level to avoid dependence on the default trust behavior of Java's TLS implementation. This is particularly true if you ever find yourself modifying the "global" Java trust store. That should never be done, since it makes Java upgrades much more troublesome.

In comparison to some of the IdP's features, the veneer here is very "thin". That is, we don't have a lot of layers of abstraction and simplification in place to hide the gory details (this may come as news, but much of the rest of the configuration is very abstracted and simplified from what it could look like). So two points: this topic will eventually be heavy on examples, and going beyond the examples is more a case of reading javadocs and finding the right settings than reading some documentation to magically impart the right answer.

General Configuration

A set of Spring factory beans are provided that understand how to build an HttpClient instance with a variety of features and settings. For technical reasons the basic caching behavior of the client is determined by selecting from among three different factory bean types:

- net.shibboleth.idp.profile.spring.relyingparty.metadata.HttpClientFactoryBean
- net.shibboleth.idp.profile.spring.relyingparty.metadata.FileCachingHttpClientFactoryBean
- net.shibboleth.idp.profile.spring.relyingparty.metadata.InMemoryCachingHttpClientFactoryBean

As you would expect, the first provides no explicit caching of results, the second caches results on disk (but **not** across restarts of the software), and the third caches results in memory. This is HTTP caching; that is, it relies on signaling between the client and web server to detect when to reuse results and supports conditional GET requests with cache control headers to redirect requests into the cache. Essentially they act much like a browser would.

Obviously if you expect the content to be fully dynamic, as with a web service, the non-caching choice is appropriate, while components that request large, infrequently-modified files will gain benefits from caching. In no case does this caching supply real high-availability resilience. The IdP tends to implement HA features itself on top of the client library to provide more control and reliability.

Built-In Clients

An instance of each of the above clients is defined by default, either for use directly or as a parent template for your own beans:

- **shibboleth.NonCachingHttpClient**
- **shibboleth.FileCachingHttpClient**
- **shibboleth.MemoryCachingHttpClient**

Many common settings are exposed as properties (mostly commented out for easy definition in *services.properties*).These properties are a mix of global settings that configure each client type and specific settings controlling the caching behavior of the individual types. This is a suitable approach if you have one use case, or are fine with all use cases sharing client behavior.

Custom Clients

If you have more advanced needs, just define your own bean that inherits from one of these, and override any settings as needed. For example, instead of relying on the global **idp.httpclient.socketTimeout** property, perhaps you want to define a special client instance with a shorter timeout:

```
<bean id="ShortTimeoutHttpClient" parent="shibboleth.NonCachingHttpClient" p:socketTimeout="PT5S" />
```

Now you have a bean name you can inject into other components that support an `httpClientRef` property that will behave differently than the defaults.

Security Configuration

A common use for advanced configuration is to control the server certificate evaluation process. While the IdP typically relies on SAML metadata for this when communicating with servers, this doesn't work for a lot of non-SAML use cases. However, much of the same machinery can be repurposed to support a variety of trust models, and we have integrated this into the HttpClient library. Another use case is to support HTTP authentication such as a name and password to access a resource.

TLS Server Authentication

By default, the HttpClient will rely on Java's built-in TLS behavior and code built into the HttpClient library, and perform basic hostname/certificate checks and will rely on the Java global trust store to determine whether to trust the certificate (or its issuer). This is where most non-Shibboleth software stops. In testing scenarios, you can turn off this checking via the `connectionDisregardTLSCertificate` property, but this should never be used in production.

For more advanced control over this process, you need to perform additional steps, and add more complex Spring wiring involving the org.opensaml.security.httpclient.HttpClientSecurityParameters class, which provides for using a TrustEngine to evaluate the server's certificate. There are a variety of TrustEngine implementations that can perform simple or advanced checks, but the critical difference is that they're contextual. That is, they can be applied to one component such that the rules it uses can be specific to that component alone and not the whole system.

A particularly useful approach is to abandon the fundamentally flawed use of commercial certificates with non-browser-facing services and use self-signed certificates evaluated on the basis of public key comparisons, much like Shibboleth does with SAML, or even certificates issued on an automated short-term basis by dedicated CAs.

If you want an HttpClient bean to use the special TLSSocketFactory we provide that supports a TrustEngine, you **MUST** provide an HttpClientSecurityParameters instance to the component using the HttpClient bean to configure the security behavior you want. Failure to do so will prevent TLS socket connections from succeeding in V3.4, but only results in warnings on earlier versions, which can lead to security mistakes.

Repeating: you can tell the HttpClient bean that you want to support a more "advanced mode" of processing, but you configure the rules not on the HttpClient bean but rather on the component using the client bean. This is because the injection of the rules you want to apply have to be added at runtime when the client gets used and not into the client's own data structures. It's a consequence of the library's design. The risk in versions of the software prior to V3.4 is that if you tell the client you want this, but don't configure the component that's using it properly, the default Java behavior is also skipped, and you get no security at all. The following explains what you should do, in detail, for older versions, because on newer versions it fails closed anyway.

TrustEngine Examples

To use a TrustEngine, you need to define an HttpClientSecurityParameters bean with a `tLSTrustEngine` property. While you can define any compatible object, the two most common cases are supported via a pair of factory beans:

- **shibboleth.StaticExplicitTrustEngine**
  - Explicitly compares a set of trusted public keys against the key in the server's certificate, with no regard for the rest of the certificate.
- **shibboleth.StaticPKIXTrustEngine**
  - Provides a full suite of PKIX processing to the server's certificate, but with dedicated trust authorities and policy rules.

The examples below are semi-complete, in that they include a portion of the "real" component that the security rules are being supplied for. Most of the components that can support this will carry both `httpClient` and `httpClientSecurityParameters` properties. Components defined in custom (non-Spring) XML syntax will usually support `httpClientRef` and `httpClientSecurityParametersRef` XML attributes to reference Spring beans of the proper type.

Unsurprisingly, the "explicit" engine is a bit simpler to use. You can provide any number of public keys and certificates via resources (file, classpath, even HTTP, though that gets pretty circular here), to drive the engine.

The common case of a rule to compare the key against a certificate loaded from a file looks like this:

Explicit key comparison via a known certificate

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticExplicitTrustEngine"
                        p:certificates="%{idp.home}/credentials/server.pem" />
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

An example that might be used to support a server rolling over to a new key:

Explicit key comparison against two keys

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticExplicitTrustEngine">
                        <property name="publicKeys">
                                <list>
                                        <value>%{idp.home}/credentials/pubkey1.pem</value>
                                        <value>%{idp.home}/credentials/pubkey2.pem</value>
                                </list>
                        </property>
                </bean>
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

The full range of PKIX options is quite complex, but for basic use cases a factory bean makes it simple. To validate the server's certificate against a fixed CA (name checking is turned off because the HttpClient is already doing this step for us):

PKIX verification with root CA

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticPKIXTrustEngine"
                        p:certificates="%{idp.home}/credentials/rootca.pem"
                        p:checkNames="false" />
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

Other PKIX options include the `verifyDepth` property to control the chain length, and the `cRLs` property to supply certificate revocation lists.

Applying a TrustEngine to HttpClient

You should start by configuring a component using the HttpClient with the HttpClientSecurityParameters wiring needed to implement your needs, as in the above examples.

Once that's in place, temporarily configure the HttpClient with the `connectionDisregardTLSCertificate` flag on, and finally just test it and let it fail and log something like the following:

```
19:04:53.364 - 127.0.0.1 - WARN [org.opensaml.security.httpclient.HttpClientSecuritySupport:98] - Configured
TLS trust engine was not used to verify server TLS credential, the appropriate socket factory was likely not
configured
19:04:53.366 - 127.0.0.1 - ERROR [net.shibboleth.idp.profile.impl.ResolveAttributes:299] - Profile Action
ResolveAttributes: Error resolving attributes
net.shibboleth.idp.attribute.resolver.ResolutionException: Data Connector 'webservice': HTTP request failed
 at net.shibboleth.idp.attribute.resolver.dc.http.impl.HTTPDataConnector.retrieveAttributes(HTTPDataConnector.
java:90)
Caused by: javax.net.ssl.SSLPeerUnverifiedException: Evaluation of server TLS credential with configured
TrustEngine was not performed
 at org.opensaml.security.httpclient.HttpClientSecuritySupport.checkTLSCredentialEvaluated
(HttpClientSecuritySupport.java:100)
```

The category and details in the ERROR will vary by component, but the WARN and SSLPeerUnverifiedException messages will consistently indicate that the component was configured to apply a TrustEngine to the connection but couldn't do so. This may happen **after** the connection is established, possibly even after data is sent and consumed, but it should happen at the end. If not, something is wrong and has to be corrected to be confident that the eventual configuration will work.

Once the warning/error occurs, only then make the final change to the HttpClient bean to fix the error by overriding the `tLSSocketFactory` property (note the weird mixed case, that's due to the usual "first character is lower case" property convention in Java beans). The IdP includes a pair of socket factory beans for this purpose, **shibboleth.SecurityEnhancedTLSSocketFactory** and **shibboleth.SecurityEnhancedTLSSocketFactoryWithClientTLS**

So the *final* step to adding advanced TLS support is, for example:

```
<bean id="SecurityEnhancedHttpClient" parent="shibboleth.NonCachingHttpClient"
        p:tLSSocketFactory-ref="shibboleth.SecurityEnhancedTLSSocketFactory" />
```

You won't generally need to configure your own instance of the socket factories, because all the interesting settings are part of the other component's configuration, via HttpClientSecurityParameters.

TLS Client Authentication

⚠ The code as it stands does not generally support TLS Renegotiation, which is most commonly encountered when using a virtual host that applies client TLS to only a subset of paths and not the host as a whole.

Configuring a component using the HttpClient with a private key and certificate for authenticating itself to a server is a simple two step process:

- Make sure the HttpClient bean's `tLSSocketFactory-ref` property points to the **shibboleth. SecurityEnhancedTLSSocketFactoryWithClientTLS** bean.
- Configure the component's injected HttpClientSecurityParameters instance's `clientTLSCredential` property with an X.509 credential.

The syntax for supplying a keypair can be essentially copied from the *credentials.xml* file that contains the more "usual" keys and certificates used by the IdP. Note that the beans defined in that file are not visible outside the RelyingPartyConfiguration so if you try to reuse them elsewhere by reference, you'll get errors. The example builds on the previous one and includes server authentication.

Client TLS example

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticExplicitTrustEngine"
                        p:certificates="%{idp.home}/credentials/server.pem" />
        </property>
        <property name="clientTLSCredential">
                <bean class="net.shibboleth.idp.profile.spring.factory.BasicX509CredentialFactoryBean"
                        p:privateKeyResource="%{idp.home}/credentials/tlsclient.key"
                        p:certificateResource="%{idp.home}/credentials/tlsclient.crt" />
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

HTTP Authentication

Currently it is less than straightforward to configure more typical HTTP credentials such as a basic-auth username and password, due to a lack of abstraction methods in our code to hide some of the gory details of the HttpClient's data model. In particular, some of the methods that need to be called take multiple parameters, which violates the bean convention for a setter. It's possible to invoke more complex methods in Spring, but it takes some extra wiring. We intend to supply some additional code for this in a future release.

Note that some of the older custom schemas such as the metadata configuration schema may support shorthand for supplying username/password credentials, and while these do work, they're deprecated in favor of the more generic `httpClientSecurityParameters-ref` syntax.

At the moment, it's fairly simple to supply a username and password that gets used unilaterally with a given component's requests. That is, it's not "scoped" to limit its use to a particular server. This implies that you have a working configuration in place to authenticate the server's certificate so that the password isn't sent inadvertently to a malicious location. An example follows (again, building on the server authentication case):

Use of Basic Authentication

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticExplicitTrustEngine"
                        p:certificates="%{idp.home}/credentials/server.pem" />
        </property>
        <property name="basicCredentials">
                <bean class="org.apache.http.auth.UsernamePasswordCredentials"
                        c:_0="webauth" c:_1="%{idp.collector.password}" />
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

The next level up in complexity is the desirable ability to limit the scope of the credentials for safety's sake. The example relies on the hostname and port of the server to scope the password. There are more advanced ways to build the AuthScope object being passed into the API such as including the Realm challenge from the server.

Basic Authentication with host/port AuthScope

```
<bean id="CustomHttpSecurity" class="org.opensaml.security.httpclient.HttpClientSecurityParameters">
        <property name="tLSTrustEngine">
                <bean parent="shibboleth.StaticExplicitTrustEngine"
                        p:certificates="%{idp.home}/credentials/server.pem" />
        </property>
</bean>

<bean id="ScopedBasicAuth" class="org.springframework.beans.factory.config.MethodInvokingBean"
                p:targetObject-ref="CustomHttpSecurity"
                p:targetMethod="setBasicCredentialsWithScope">
        <property name="arguments">
                <list>
                        <bean class="org.apache.http.auth.UsernamePasswordCredentials"
                                c:_0="webauth" c:_1="%{idp.collector.password}" />
                        <bean class="org.apache.http.auth.AuthScope"
                                c:_0="log.example.org" c:_1="443" />
                </list>
        </property>
</bean>

<!-- Sample feature we're actually trying to use, which we inject custom rules into. -->
<bean id="PushReporter" parent="shibboleth.metrics.HTTPReporter" c:name="MyCollector"
        p:httpClient-ref="CustomHttpClient"
        p:httpClientSecurityParameters-ref="CustomHttpSecurity"
        p:collectorURL="https://log.example.org/cgi-bin/collector.cgi" />
```

A more advanced example would be to configure multiple sets of credentials for different servers, assuming a component that potentially contacts different servers. Since this is not a common case with any of our components, it's not likely to be needed much.

Reference

Beans

Name	Type	Description
shibboleth.NonCachingHttpClient	HttpClientFactoryBean	Factory bean for creating non-caching HTTPClient
shibboleth.FileCachingHttpClient	FileCachingHttpClientFactoryBean	Factory bean for creating file-based-caching HTTPClient
shibboleth.MemoryCachingHttpClient	InMemoryCachingHttpClientFactoryBean	Factory bean for creating in-memory-caching HTTPClient
shibboleth.StaticExplicitTrustEngine 3.3	StaticExplicitKeyFactoryBean	Factory bean for creating ExplicitKeyTrustEngine
shibboleth.StaticPKIXTrustEngine 3.3	StaticPKIXFactoryBean	Factory bean for creating PKIXX509CredentialTrustEngine
shibboleth.SecurityEnhancedTLSSocketFactory 3.2	SecurityEnhancedTLSSocketFactory	Socket factory that supports HttpClientSecurityParameters-aware components
shibboleth.SecurityEnhancedTLSSocketFactoryWithClientTLS 3.3	SecurityEnhancedTLSSocketFactory	Client-TLS-capable socket factory that supports HttpClientSecurityParameters-aware components
shibboleth.SecurityEnhancedTLSSocketFactoryWithClientTLSOnly 3.4	SecurityEnhancedTLSSocketFactory	Client-TLS-capable socket factory that supports HttpClientSecurityParameters-aware components but does not accept a pluggable TrustEngine

Properties

Name	Type	Default	Description

idp.httpclient.useSecurityEnhancedTLSSocketFactory [3.2]	Boolean	false	If true, causes the default clients to be injected with a special socket factory that supports advanced TLS features (requires substantial additional configuration)
idp.httpclient.connectionDisregardTLSCertificate	Boolean	false	If the previous property is false, this allows the default TLS behavior of the client to ignore the TLS server certificate entirely (use with obvious caution, typically only while testing)
idp.httpclient.connectionRequestTimeout	Duration	PT1M (one min)	TIme to wait for a connection to be returned from the pool (can be 0 for no imposed value)
idp.httpclient.connectionTimeout	Duration	PT1M (one min)	TIme to wait for a connection to be established (can be 0 for no imposed value)
idp.httpclient.socketTimeout	Duration	PT1M (one min)	Time to allow between packets on a connection (can be 0 for no imposed value)
idp.httpclient.maxConnectionsTotal	Integer	100	Caps the number of simultaneous connections created by the pooling connection manager
idp.httpclient.maxConnectionsPerRoute	Integer	100	Caps the number of simultaneous connections per route created by the pooling connection manager
idp.httpclient.memorycaching.maxCacheEntries	Integer	50	Size of the in-memory result cache
idp.httpclient.memorycaching.maxCacheEntrySize	Long	1048576 (1MB)	Largest size to allow for an in-memory cache entry
idp.httpclient.filecaching.maxCacheEntries	Integer	100	Size of the on-disk result cache
idp.httpclient.filecaching.maxCacheEntrySize	Long	10485760 (10MB)	Largest sze to allow for an on-disk cache entry
idp.httpclient.filecaching.cacheDirectory	Local directory		Location of on-disk cache